版权注意事项：
1、书籍版权归作者和出版社所有
2、本PDF仅限用于个人获取知识，进行私底下的知识交流
3、PDF获得者不得在互联网上以任何目的进行传播
4、如觉得书籍内容很赞，请购买正版实体书，支持作者
5、请于下载PDF后24小时内删除本PDF。

artima

异步图书
www.epubit.com.cn

SCALA

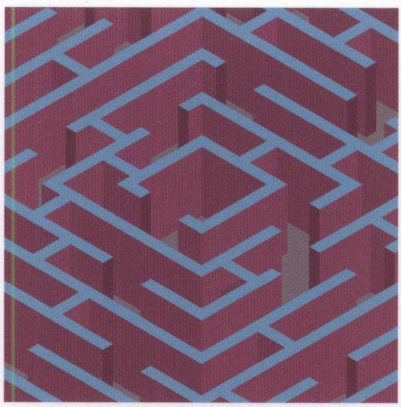
深入理解 Scala 的有趣途径

Scala 谜题

[美] Andrew Phillips Nermin Šerifović 著

包春霞 冷钰冰 译

PUZZLERS



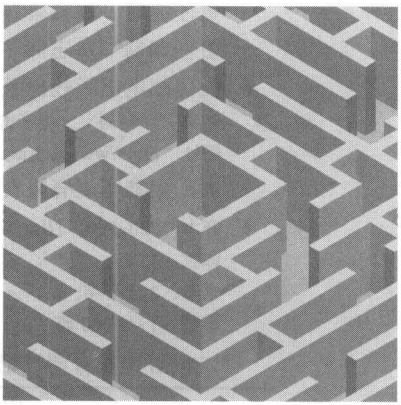
中国工信出版集团

人民邮电出版社
POSTS & TELECOM PRESS

Scala 谜题

[美] Andrew Phillips Nermin Šerifović 著

包春霞 冷钰冰 译



人 民 邮 电 出 版 社

北 京

图书在版编目（CIP）数据

Scala谜题 / (美) 菲利普斯 (Andrew Phillips) , (美) 萨尔法维克著 ; 包春霞, 冷钰冰译. -- 北京 : 人民邮电出版社, 2017.11
ISBN 978-7-115-46007-3

Ⅰ. ①S… Ⅱ. ①菲… ②萨… ③包… ④冷… Ⅲ. ①JAVA语言－程序设计 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字(2017)第190917号

版权声明

Simplified Chinese translation copyright ©2017 by Posts and Telecommunications Press
ALL RIGHTS RESERVED
Scala Puzzlers by Andrew Phillips and Nermin Šerifović ISBN 9780981531670
Copyright © 2016 by Artima, Inc.

本书中文简体版由 **Artima, Inc.**授权人民邮电出版社出版。未经出版者书面许可，对本书的任何部分不得以任何方式或任何手段复制和传播。
版权所有，侵权必究。

◆ 著　[美] Andrew Phillips　Nermin Šerifović
译　包春霞　冷钰冰
责任编辑　陈冀康
责任印制　焦志炜

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路 11 号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
大厂聚鑫印刷有限责任公司印刷

◆ 开本：720×960　1/16
印张：12.75
字数：225 千字　2017 年 11 月第 1 版
印数：1－2 400 册　2017 年 11 月河北第 1 次印刷

著作权合同登记号　图字：01-2016-0525 号

定价：49.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字 20170147 号

内容提要

Scala 是一种多范式的编程语言，其设计初衷是要整合面向对象编程和函数式编程的各种特性。

本书整合了众多典型的Scala代码示例，深入解密Scala。书中不仅介绍了Scala语言，还介绍了编译器。本书通过有趣的方式带领读者学习并深入理解和掌握Scala。全书共有 36 个谜题，每一个谜题都可以丰富读者的知识，并能够让读者更深入地了解 Scala。

本书适合于对 Scala 感兴趣的开发者、对 JVM 平台上的语言以及函数式编程感兴趣的程序员阅读。

序　言

谜题很有趣。我还记得第一次遇到 Java 语言的“谜题”的情景。Neal Gafter 给我看了他和 Josh Bloch 刚收集的一道 Java 谜题。那时候，Neal 正在维护我编写的 javac 编译器。这道题太难了，我猜错了几次，让 Neal 乐坏了。

很高兴现在有一本书能在 Scala 中延续谜题的传统。本书将为读者展示 36 个谜题，之所以称为谜题，是因为它们会产生令人意外的结果，有特性间的交互作用或者有不同于表面的代码运行结果。这些谜题是从 Scala 社区几年来广泛的输入中收集而来的。

Andrew 和 Nermin 提炼出每个谜题的本质，使其易于理解。在享受了从一系列选项中选出正确答案的乐趣之后，每道题后面会告诉你为什么会这样，什么原因导致了这个令人意外的结果？这正是本书真正的亮点，因为它对观察到的程序行为的深层原理进行了清晰地讲解。我特别喜欢本书的一点是这些讲解总是给你带来全新的视角。它们不仅告诉你令人意外的行为的趣事，而且集 Scala 深入讲解于一体。这样，谜题就会有助于你更深入地理解这门语言。

希望你能像我一样觉得阅读本书并试着解决这些谜题很有趣。是的，有些谜题我还不能解决。

Martin Odersky

Scala 之父

2014.5.26

自　序

当开始学习一门新的编程语言时，最初困扰你的只是缺少这门语言的知识。随着你对这门语言的经验和能力的增长，让你头疼的时刻就逐渐变成了系统层面的问题，例如，依赖冲突或很难重现的竞争条件。让人真正喜欢 Josh Bloch 和 Neal Gafter 的 Java 谜题的一点是：它让我们在日常工作和最后期限之外参与 Java。谜题不仅触发了我们解决问题的欲望，而且帮助我们纠正并加深了对这门语言的理解。

开始学习 Scala 不久，我们就认识到收集一套类似的谜题可能会真的很有用——由具有时代精神的社区来建立并维护这些谜题是再好不过的了。因而，Scala 谜题网站 scalapuzzlers.com 就诞生了。

受所收到的积极反馈的鼓励，我们决定将这些谜题收集并扩展成本书。我们的目标是：更加详细地解释这些谜题的行为，并讨论潜在的含义以及可能的权变措施。我们力求将每个谜题当作一个机会，来展示这门语言的不同寻常之处或突出一种常用语言特性里鲜为人知的方面。

详细地研究并解释每个谜题有很多乐趣，也有很多学习经验。希望你像我们一样，能从本书中得到很多收获，包括知识和娱乐。

最后，如果你遇到 Scala 代码的迷惑不解之处，请通过 Scala 谜题网站 scalapuzzlers.com 提交给我们。谜题社区期待你的加入。

Andrew Phillips，Nermin Šerifović

马萨诸塞州波士顿市

2014.5.28

致　谢

感谢每一位参与者，无论是提供了谜题代码，还是写了注释或建议，或者仅仅是传播了文字。感谢你们使本书的编写和 Scala 谜题网站 scalapuzzlers.com 的制作得以实现。

真挚地感谢我们的编辑 Jessica Kerr、Bill Venners 和 Theresa Gonzalez，感谢他们的探索和对本书的全面评审，感谢 Darlene Wallach 和 George Berger 排版并采用发布系统来处理这本书。

还要特别感谢 Scala 谜题 scalapuzzlers.com 网站的所有贡献者：Dominik Gruntz、A. P. Marki、Simon Schäfer、Konstantine Golikov、Seth Tisue、Daniel C. Sobral、Luc Bourlier、Vassil Dichev 和 Andraž Bajt。你们的贡献为本书提供了基础和灵感。

也要对那些指出错误和提出改进建议的读者表示感谢：Cay Horstmann、Harish Hurchurn、Marcin Kubala、Edward G. Prentice，Alex Varju。特别感谢 Dominik Gruntz 的全面评审并提出许多有用的建议。

献　　辞

我的母亲 Karin 如涓涓细流般美妙的文笔熏陶我、影响我写成本书；我的两个耐心的朋友 Libby 和 Bill Venners，她们一直在用 Scalac 编译程序，要不是她们，我就不会认识 Scala。

——A.P.

我出色的妻子 Džana 毫无保留的支持，帮我完成了另一个副项目。我的两个调皮的儿子 Immy 和 Rayan，他们很好奇我是否在写一本“真正”的、以“从前”开始，以“结束”结尾的书。我敬爱的父母，Nad 和 Sabrija，他们在孩子的教育上投入大量的时间。

——N.Š.

前　言

让代码做我们希望它做的事，是作为一个开发者的基本目的。所以没有什么比我们自认为理解的一段代码表现出与我们期望相反的行为更迷人、更重要的了。

本书汇集了一些 Scala 的例子。这不仅是以一种寓教于乐的方式来更好地理解这门语言，而且帮助你认识许多反直观的雷区和陷阱，防止它们在生产系统中干扰到你。

本书主要内容

全书共有 36 章，具体内容如下。

- 第 1 章“使用占位符”，本章中，我们通过引入占位符语法从函数调用中删除了一些陈词滥调。
- 第 2 章“初始化变量”，本章中，我们检查了 Scala 里给多个变量赋值的盛衰起伏。
- 第 3 章“成员声明的位置”，本章我们审视了类成员的周边环境对初始化的影响。
- 第 4 章“继承”，本章我们通过类初始化顺序跟踪几个抽象和重载值。
- 第 5 章“集合操作”，这一章我们试图计数几个集合而不用太担心它们确切的类型。
- 第 6 章“参数类型”，本章我们通过定义两种特定类型的帮助函数的通用版本尽量写出好的函数式程序。
- 第 7 章“闭包”，本章对几个延迟的访问器函数上的索引数据序列略窥一二。
- 第 8 章“Map 表达式”，本章我们用 for 表达式并通过 map 的应用来转换一些字母汤。
- 第 9 章“循环引用变量”，本章我们定义并随机地访问一对相互引用的变量。

- 第 10 章“等式的例子”，本章我们在 case class 的 hashCode 方法中混进调试代码。
- 第 11 章“lazy val”，本章我们试图初始化一个随环境而变化的 lazy 值。
- 第 12 章“集合的迭代顺序”，本章我们试图按值的顺序输出罗马数字。
- 第 13 章“自引用”，本章我们会查看 Scala 如何处理自引用变量。
- 第 14 章“Return 语句”，本章我们将体验程序流控制。
- 第 15 章“偏函数中的_”，本章我们调用两个偏应用函数作为计数器。
- 第 16 章“多参数列表”，本章我们给偏函数提供不同数量的参数。
- 第 17 章“隐式参数”，本章我们从一个有隐式参数的方法创建一个偏应用函数。
- 第 18 章“重载”，本章我们定义并调用参数数量可变的重载方法。
- 第 19 章“命名参数和缺省参数”，本章我们分别调用一个指定和没有指定参数名的方法。
- 第 20 章“正则表达式”，本章我们从正则匹配中删除一些调试代码。
- 第 21 章“填充”，本章我们每次用一个“*”字符按希望的长度填充一个字符串。
- 第 22 章“投影”，本章我们将一个从调用 Java 代码（gulp!）返回的 map 投影到一个 Scala 类型的 map。
- 第 23 章“构造器参数”，本章我们实例化一个类的多个实例，这个类的构造器是由按名字传递的参数指定的。
- 第 24 章“Double .NaN”，本章我们对几个 double 数组进行排序。
- 第 25 章“getOrElse”，本章我们组合两个列表，当试图从结果中抽取一个元素时就有些令人迷惑了。
- 第 26 章“Any Args”，本章我们重构一个方法，当调用方法的代码忘记改变时，用多个参数列表的方式调用重构的方法依然可以成功。
- 第 27 章“null”，本章我们对一个从调用 Java 代码返回的字符串应用模式匹配。
- 第 28 章“AnyVal”，本章我们初始化一个其特定的类型在子类中声明的 AnyVal 子类型。

- 第 29 章“隐式变量”，本章我们通过导入附加的隐式变量的方式在程序中间创建一个“测试模式”上下文。
- 第 30 章“显式声明类型”，本章我们定义两种隐式转换，让编译器去弄清楚它们的返回类型。
- 第 31 章“View”，本章我们优化一个只能在 map 数据项的值上操作的 map 调用。
- 第 32 章“toSet”，本章我们在增加附加的元素之前将一个列表转换成一个集合。
- 第 33 章“缺省值”，本章我们使用两种不同的方式为 map 数据项指定缺省值。
- 第 34 章“关于 Main”，本章我们对一个用显式 main 方法的“保守”对象进行重构，把它修改成继承 App 特质的方法。
- 第 35 章“列表”，本章我们使用圆括号和大括号定义一个匿名函数。
- 第 36 章“计算集合的大小”，本章我们重构一个计数方法让它可以处理通用的集合类型。

如何阅读本书

本书中的谜题并非按特定顺序编写。你可以随机打开一个谜题阅读，会与从头到尾地读一样轻松。

如果你对 Scala 语言的特定方面感兴趣并且在寻找相关的谜题，本书最后的主题索引正是为你准备的。我们尽量根据读者要探索的主题对谜题进行了分类。

谜题中所有的代码例子都要用 2.11 Scala REPL[①]解释执行，最近的一些变更，例如一些不支持的写法会让程序行为与 Scala 2.10.X 版本稍微不同，我们已经增加了注释或注脚。

尽管本书经过了严格评审，但错误还是不可避免的。如果你发现任何错误，请通过 http://booksites.artima.com/scalapuzzlers/errata 告诉错误所在的页码。

---

① 从命令行输入“Scala”来启动 REPL（读—评估—打印—循环，Read-Evaluate-Print-Loop，REPL）。

电子书

本书同时提供印刷版书籍和 PDF 格式电子书。电子书并不是本书印刷版的简单的电子版复制。在保持与印刷版内容相同的同时，电子书还经过仔细设计和优化以适应用计算机屏幕阅读。

首先要注意的是，电子书里大部分引用是超链接的形式。如果你选择进入某章、某个图形或词汇条目的超链接，PDF 查看器会立即将你带到选择的条目而不必翻来翻去地寻找。

此外，电子书的每页底部是一些导航链接。“封面”“概述”“目录”链接将你带到本书的开头；“索引”链接将你带到本书最后的索引部分；“讨论”链接将你带到论坛，在那里你可以与其他读者、作者和更大的 Scala 社区讨论问题。如果你发现一个拼写错误或者某些你认为能解释得更好的地方，请单击“建议”链接进入在线 Web 应用，在这里你可以给作者提供反馈。

尽管电子版与印刷版有相同的页面，但我们已经删除了电子书的空白页，并对剩下的页面重新编排了页码。各页的页码不同以便当你只打印本书的一部分时很容易决定 PDF 的页数。电子书的页码已经严格按照 PDF 阅读器计数方式进行了编排。

排版约定

一个术语首次使用的时候是斜体字。小的代码样例在同一行里用单间隔字体，例如 `x + 1`。较大的代码样例放在单行间隔的引用块中，如下所示：

```
def hello() {
  println("Hello, world!")
}
```

当出现交互式 Shell 时，Shell 返回结果用亮字体显示：

```
scala> 3 + 4
res0: Int = 7
```

谜题概览

目　录

第 1 章

使用占位符

Scala 特别强调要书写简单、简洁的代码。匿名函数的语法 `arg => expr`，使它很容易用最小模板构建函数字面量，甚至函数由多个语句组成时也一样可以。

用有自解释参数的函数还可以做得更好，而且还可以用占位符语法。占位符语法可以省去参数声明。例如：

```
List(1,2).map { i => i + 1 }
```

用占位符语法，就变成：

```
List(1,2).map { _ + 1 }
```

以下两个语句是等价的：

```
scala> List(1,2).map { i => i + 1 }
res1: List[Int] = List(2,3)

scala> List(1,2).map { _ + 1 }
res0: List[Int] = List(2,3)
```

如果你给以上简单的例子增加调试语句会如何呢？这有助于你理解函数是什么时候应用的。让我们看看在 REPL 中执行以下代码会是什么结果。

```
List(1,2).map { i => println("Hi");i + 1 }
List(1,2).map { println("Hi"); _ + 1 }
```

可能的结果

1. 打印出：

```
Hi
```

```
List[Int] = List(2,3)
Hi
List[Int] = List(2,3)
```

2. 打印出：

```
Hi
Hi
List[Int] = List(2,3)
Hi
Hi
List[Int] = List(2,3)
```

3. 打印出：

```
Hi
Hi
List[Int] = List(2,3)
Hi
List[Int] = List(2,3)
```

4. 第一个语句打印出：

```
Hi
Hi
List[Int] = List(2,3)
```

第二个语句编译失败。

解释

你无需关心编译错误，因为代码编译没有问题。然而程序并没有表现出你期望的结果。正确的答案是 3：

```
scala> List(1, 2).map { i => println("Hi"); i + 1 }
Hi
Hi
res23: List[Int] = List(2,3)

scala> List(1, 2).map { println("Hi"); _ + 1 }
Hi
res25: List[Int] = List(2,3)
```

怎么会是这样呢？如果这个有显式参数的函数打印“Hi”两次，因为它会对列表中的每个元素都调用一次函数，那么为什么使用占位符语法的函数不能有同样的结果呢？

因为匿名函数常常被当作参数传递，在代码中往往会看到它们在花括号{ ... }里，就很容易认为这些花括号表示一个匿名函数。但是，实际上它们只是界定了一个块表达式，一个或多个表达式最后决定了这个块的结果。

两个代码块的解析方式决定了它们有不同的行为。第一个语句{ i => println("Hi"); i + 1 }被当成一个 arg => expr 形式的函数字面量表达式，这里的表达式是块 println("Hi"); i + 1。因为 println 语句是函数体的一部分，所以每次调用函数就要执行一次。

```
scala> val printAndAddOne =
         (i: Int) => { println("Hi"); i + 1 }
printAndAddOne: Int => Int = <function1>

scala> List(1, 2).map(printAndAddOne)
Hi
Hi
res29: List[Int] = List(2, 3)
```

第二个表达式中，代码块被认为是 println("Hi")和_+ 1 两个表达式。当这个代码块执行的时候，将最后一个表达式（便利性所需的函数类型，Int => Int）传递给 map。其中的 println 语句不是函数体的一部分，它是在 map 的参数评估时被调用的，而不是作为 map 的一部分执行。

```
scala> val printAndReturnAFunc =
         { println("Hi"); (_: Int) + 1 }
Hi
printAndReturnAFunc: Int => Int = <function1>

scala> List(1, 2).map(printAndReturnAFunc)
res30: List[Int] = List(2, 3)
```

讨论

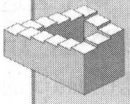
这里学到的关键一点是：用占位符语法定义的匿名函数的范围只延伸到含有下划线（_）的表达式。这不同于常规的匿名函数，常规的匿名函数的函数体是包含从箭头标识符（=>）一直到代码块结束的所有代码。请看下面这个例子：

```
scala> val regularFunc =
         { a: Any => println("foo"); println(a); "baz" }
regularFunc: Any => String = <function1>
```

```
scala> regularFunc("hello")
foo
hello
res42: String = baz
```

而使用占位符语法的函数被封闭在它自己的代码块里。例如，下面两个函数是等价的：

```
scala> val anonymousFunc =
         { println("foo"); println(_: Any); "baz" }
foo
anonymousFunc: String = baz

scala> val confinedFunc =
         { println("foo"); { a: Any => println(a) }; "baz" }
foo
confinedFunc: String = baz
```



Scala 鼓励简洁的代码，但太简洁时就会出现这样的情况。使用占位符语法时一定要注意由它所创建的函数范围。

第 2 章

初始化变量

Scala 提供了几种便利方法可以初始化多个变量。有时候，这些方法会带来意想不到的结果。

在 REPL 中执行以下代码会是什么结果呢？

```
var MONTH = 12; var DAY = 24
var (HOUR, MINUTE, SECOND) = (12, 0, 0)
```

可能的结果

1. 打印出：

```
MONTH: Int = 12
DAY: Int = 24
HOUR: Int = 12
MINUTE: Int = 0
SECOND: Int = 0
```

2. 两个语句都编译失败。

3. 第一个语句打印出：

```
MONTH: Int = 12
DAY: Int = 24
```

第二个语句抛出运行时异常。

4. 第一个语句打印出：

```
MONTH: Int = 12
DAY: Int = 24
```

第二个语句编译失败。

解释

你可能会想起关于大写变量和常数值的什么信息，怀疑这两个语句真能有一个会编译通过吗？如所发生的那样，第一行编译通过，第二行编译失败。正确的答案是 4：

```
scala> var MONTH = 12; var DAY = 24
MONTH: Int = 12
DAY: Int = 24

scala> var (HOUR, MINUTE, SECOND) = (12, 0, 0)
<console>:11: error: not found: value HOUR
       var (HOUR, MINUTE, SECOND) = (12, 0, 0)
            ^


<console>:11: error: not found: value MINUTE
       var (HOUR, MINUTE, SECOND) = (12, 0, 0)
                  ^


<console>:11: error: not found: value SECOND
       var (HOUR, MINUTE, SECOND) = (12, 0, 0)
```

Scala 会让你对简单的单值赋值的 val 和 var 使用大写变量名，如例子中的 MONTH 和 DAY。

当第二条语句用大写变量名技巧性地给多变量赋值的时候，这种技巧就引起了程序例外，因为多变量赋值是基于模式匹配的，而在一个模式匹配中，以大写字母开头的变量有着特别的含义：它们是静态标识符。

静态标识符是用来匹配常量的：

```
scala> final val TheAnswer = 42
scala> def checkGuess(guess: Int) = guess match {
         case TheAnswer => "Your guess is correct"
         case _ => "Try again"
       }
scala> checkGuess(21)
res8: String = Try again
```

```
scala> checkGuess(42)
res9: String = Your guess is correct
```

相反，小写变量定义的是变量的模式，这个模式会给变量赋值。

```
scala> var (hour, minute, second) = (12, 0, 0)
hour: Int = 12
minute: Int = 0
second: Int = 0
```

在我们的代码例子中，原意并不是对变量赋值，而是要匹配常量值。

讨论

如果你正在试图使用大写变量名，在极其偶然的情况下，恰巧匹配了范围内的值（在大型程序中常用的名字），模式匹配能编译成功，但成功还是失败取决于是否有值与之匹配：

```
val HOUR = 12; val MINUTE, SECOND = 0;

scala> var (HOUR, MINUTE, SECOND) = (12, 0, 0)

val HOUR = 13; val MINUTE, SECOND = 0;

scala> var (HOUR, MINUTE, SECOND) = (12, 0, 0)
scala.MatchError: (12,0,0) (of class scala.Tuple3)
  ...
```

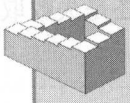
注意，即便是第一种情况匹配成功，但实际上也没有变量被赋值：按照定义，在模式匹配时绝不会给静态标识符赋值。总之，最好什么都别发生，否则就会得到一个运行时例外。而这两种结果都不是我们想要的。

把小写变量放在单引号内时也能当做静态标识符使用。但要把它当作常量对待，这种情况就必须是 val。

```
final val theAnswer = 42
def checkGuess(guess: Int) = guess match {
  case `theAnswer` => "Your guess is correct"
  case _ => "Try again"
}

scala> checkGuess(42)
res0: String = Your guess is correct
```

```
var theAnswer: Int = 42 // not a val, and not final either

scala> def checkGuess(guess: Int) = guess match {
         case `theAnswer` => "Your guess is correct"
         case _ => "Try again"
       }
<console>:9: error: stable identifier required, but
  theAnswer found.
         case `theAnswer` => "Your guess is correct"
```

用大写名的 var 是没有考虑 Scala 的最佳实践，一般不太可能会遇到这样的意外：var 用小写（最好完全避免使用大写!），常量用大写。正如《Scala 语言规范》所描述的那样，还应将常量声明为 final[①]，这可以阻止子类重载它们，且当编译器能内联这些常量时还能得到额外的性能优点。



仅对常量使用大写的变量名。

① Odersky，《Scala 语言规范》，4.1 节。[Ode14]

第3章

成员声明的位置

在许多面向对象的语言中，常常在类构造器中接受参数，目的是将参数赋值给类成员。

```
class MyClass(param1, param2, ...) {
  val member1 = param1
  val member2 = param2
  ...
}
```

崇尚简洁代码的 Scala 可以通过一次声明多个成员和构造器参数来避免这种冗余。

```
class MyClass(val member1, val member2, ...) {
  ...
}
```

执行下面的代码会是什么结果呢？

```
trait A {
  val audience: String
  println("Hello " + audience)
}

class BMember(a: String = "World") extends A {
  val audience = a
  println("I repeat: Hello " + audience)
}

class BConstructor(val audience: String = "World") extends A {
  println("I repeat: Hello " + audience)
}

new BMember("Readers")
new BConstructor("Readers")
```

可能的结果

1. 打印出：

```
Hello Readers
I repeat: Hello Readers
Hello Readers
I repeat: Hello Readers
```

2. 打印出：

```
Hello World
I repeat: Hello Readers
Hello World
I repeat: Hello Readers
```

3. 打印出：

```
Hello null
I repeat: Hello Readers
Hello Readers
I repeat: Hello Readers
```

4. 打印出：

```
Hello null
I repeat: Hello Readers
Hello null
I repeat: Hello Readers
```

解释

这里的关键问题是，明确地把"reader"值赋给 audience 是什么时候变得可见的，你可能也想知道：缺省值"World"是否参与又是如何参与的呢？将 audience 的成员声明移进构造器参数列表的小小优化确定无疑没有影响吗？非也——正确答案是 3。

```
scala> new BMember("Readers")
Hello null
I repeat: Hello Readers
res3: BMember = BMember@1aa6f6eb
```

```
scala> new BConstructor("Readers")
Hello Readers
I repeat: Hello Readers
res4: BConstructor = BConstructor@64b6603a
```

换句话说，如果其成员是在 B 的构造器参数中声明的而不是构造体中声明的，那么 A 中 audience 的值就会不同。

为了理解成员声明在类体中和在构造器参数列表中的不同，你需要检查 Scala 的 class 初始化序列。让我们再看看类声明：

```
class BMember(a: String = "World") extends A {
  ...
}

class BConstructor(val audience: String = "World") extends A {
  ...
}
```

这两个类声明都是第一种形式[①]：

```
class c(param1) extends superclass { statements }
```

根据语言规范[②]，`new BMember("Readers")` 和 `new Bconstructor("Readers")` 的初始化序列是：

1. 评估`"Reader"`参数。在这个例子中，没有什么要做的，但如果参数是表达式（例如，`"readers".capitalize`）就会先评估。

2. 要构造的类由评估模板初始化[③④]。

```
superclass { statements }
```

a）首先是超类构造器 A；

b）然后是子类程序体 `BMember` 或 `BConstructor` 中的语句序列。

注意，这里我们省略了关于 trait 等的详细信息（本例中没有用到的特性）。`BMember` 的例子中，第一步将`"Readers"`分配给构造器参数。当调用 A 的构造器参数时，`audience` 还没有初始化，所以就打印出缺省的字符串值 `null`。

---

① Odersky，《Scala 语言规范》，5.3 节。[Ode14]

② Odersky，《Scala 语言规范》，5.1 节。[Ode14]

③ Odersky，《Scala 语言规范》，5.1 节。[Ode14]

④ 模板是一个类、特质或单个对象定义的程序体。它定义了一个类、特质或对象的类型信号、行为和初始化状态。

只有当 BMember 程序体的语句序列执行的时候，才会将"Readers"分配给 audience，然后打印出来。

BConstructor 的情况则不同：这里"Readers"被评估并立即赋值给 audience 作为构造器参数评估的一部分。在调用 A 的构造器时 audience 的值已经是"Readers"了。

讨论

通常，BConstructor 中的模式是首选的，因为它的行为更少可能会引起意外。超类中声明的 val 绝不会存在于非初始化的状态。

通过使用一个早期字段定义[①]从句不用在构造器参数列表中声明 audience 就可以实现相同的结果。这允许你在构造器参数上执行附加的计算（例如，规范化字符串参数的大小写）或者用正确的初始化值创建匿名类：

```
class BEarlyDef(a: String = "World") extends {
  val audience = a
} with A {
  println("I repeat: Hello " + audience)
}

scala> new BEarlyDef("Readers")
Hello Readers
I repeat: Hello Readers
res7: BEarlyDef = BEarlyDef@44c93da7

scala> new {
         val audience = "Readers"
       } with A {
         println("I repeat: Hello " + audience)
       }
Hello Readers
I repeat: Hello Readers
res0: A = anon1@71e16512
```

早期定义在超类构造器调用之前定义成员并给它赋值。根据可应用于本例的语言规范部分[②]，初始化序列是通过评估模板初始化类的：

---

① Odersky，《Scala 语言规范》，5.1.6 节。[Ode14]

② Odersky，《Scala 语言规范》，5.1.1 节、5.1 节和 5.1.6 节。[Ode14]

1．首先，按照定义的顺序早定义；

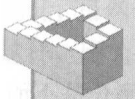
2．然后，超类构造器；

3．最后，缺省构造器中的语句。

简而言之，超类和超特质初始化代码的执行是在参数评估和早期字段定义之后，实例化类和特质的初始化语句之前。直接超类和混进特质是当它们出现在 class、trait、object 定义中时按从左到右的顺序初始化。

下面是将所有这些整合在一起的一个扩展的代码例子：

```
trait A {
  val audience: String
  println("Hello " + audience)
}

trait AfterA {
  val introduction: String
  println(introduction)
}

class BEvery(val audience: String) extends {
  val introduction =
    { println("Evaluating early def"); "Are you there?" }
} with A with AfterA {
  println("I repeat: Hello " + audience)
}

scala> new BEvery({ println("Evaluating param"); "Readers" })
Evaluating param
Evaluating early def
Hello Readers
Are you there?
I repeat: Hello Readers
res3: BEvery = BEvery@6bcc2569
```



考虑在类或对象体的括号后（这是主构造器）按照从左到右的声明顺序插入超类构造器和超特质初始化程序。

第 4 章

继承

Scala 支持面向对象的编程概念，继承是它的一个很重要的特征。继承通常对父类和特质中定义的缺省值的重载很有用。当增加多级继承时事情变得更加有趣，例如下面这段程序。

让我们看看它会打印出什么。

```
trait A {
  val foo: Int
  val bar = 10
  println("In A: foo: " + foo + ", bar: " + bar)
}

class B extends A {
  val foo: Int = 25
  println("In B: foo: " + foo + ", bar: " + bar)
}

class C extends B {
  override val bar = 99
  println("In C: foo: " + foo + ", bar: " + bar)
}

new C()
```

可能的结果

1. 打印出：

```
In A: foo: 0, bar: 0
In B: foo: 25, bar: 0
In C: foo: 25, bar: 99
```

2．打印出：

```
In A: foo: 0, bar: 10
In B: foo: 25, bar: 10
In C: foo: 25, bar: 99
```

3．打印出：

```
In A: foo: 0, bar: 0
In B: foo: 25, bar: 99
In C: foo: 25, bar: 99
```

4．打印出：

```
In A: foo: 25, bar: 99
In B: foo: 25, bar: 99
In C: foo: 25, bar: 99
```

解释

正确的答案是 1。要理解为什么会是这样的结果，还需要深入观察程序每一步是如何执行的。首先，你应该记得每个 Scala 类都有一个非显式定义但与类定义交互的主构造器[①]。类定义中的所有语句形成了主构造器的程序体，包括字段定义（顺便说一下，这就是为什么 Scala 构造器类字段和值之间并没有本质不同的原因）。因此，在特质 A、类 B 和类 C 中的所有代码都属于构造器程序体。

以下规则控制 val[②]的初始化和重载行为：

1．超类会在子类之前初始化；

2．按照声明的顺序对成员初始化；

3．当一个 val 被重载时，只能初始化一次；

4．与抽象 val 类似，重载的 val 在超类构造期间会有一个缺省的初始值。

因此，虽然表面上看在特质 `A` 和类 `B` 中给 `bar` 分配了一个初始值，实际上却不是这样，因为类 `C` 中重载了 `bar`。这意味着特质 `A` 构造时，`bar` 的缺省初始值是 0 而不是分配的值 10。实际上是初始化顺序引起了这个问题，特质 `A` 给 `bar`

① Odersky，《Scala 语言规范》，5.3 节。[Ode14]

② “为什么我的抽象或重载的 val 是 null 呢？”[为什么]

赋值 10 是完全不可见的，因为类 C 中重载了 bar，并将 bar 初始化为 99。foo 的值也类似，因为在类 B 中给 foo 赋值为非缺省值，所以 foo 的值在类 A 中是 0，随后在 B 和 C 中变成 25。

这个问题可以用抽象字段来证明，使用一个声明之后还没在子类中初始化的抽象字段就会出现这样的问题。一般而言，在类构造（包括非抽象字段）期间初始化的所有构造器及其所依赖的抽象字段很容易发生初始化顺序问题。

讨论

Scala 从 Java 中继承了初始化顺序规则。Java 确保首先初始化超类，这样就可以从子类构造器中安全使用超类字段，确保正确地初始化字段。特质会被编译成接口和具体的（非抽象的）类，所以也可应用同样的规则。

缺省初始值

Scala 给记录赋的缺省初始值为：

- Byte、Short 和 Int 类型 val 的初始值是 0
- Int、Long、Float 和 Double 类型 val 的初始值分别是 0L、0.0f 和 0.0d
- Char 类型 val 的初始值是'\0'
- Boolean 类型 val 的初始值是 false
- Unit 的初始值是 ()
- 所有其他类型的初始值是 null

你可能会问，当使用一个还没有初始化成非缺省值[①]的抽象字段时，编译器是否能用某种方式发出警告呢。可惜，缺省情况下编译器不能报出这种有关未初始化值的警告（只有测试才能捕获这样的信息）。有个高级编辑选项可以用来检查未初始化值：

-Xcheckinit 包装字段访问器对访问未初始化抽象字段抛出一个异常

① 例如，C 编译器以“在这个函数中可能使用了还未初始化的 X”这种方式发出警告。

这个选项会给所有潜在的未初始化字段访问增加一个包装器，当要访问未初始化字段时会抛出异常而不是直接使用缺省值。运行时检查字段访问器这个选项会给程序增加极大的负载，所以不推荐在生产系统代码中使用。

如果在启动 Scala REPL 会话时带上-Xcheckinit 标记，在执行 **new** C() 时就会抛出以下异常：

```
scala> new C()
scala.UninitializedFieldError: Uninitialized field:
    <console>: 10
  at C.bar(<console>:10)
  at A$class.$init$(<console>:10)
  ...
```

如果想在自动 build 中打开**-Xcheckinit** 标记以便及早发现这样的问题，这是一个好的做法。

既然已经认识到了这种问题的原因，那么该如何解决呢？后面几节会介绍几种解决方法。

解决方法

用定义的方法

一种方法是将 bar 声明为 def，而不是 val。下面这段程序就能得到你期望的运行结果：

```
trait A {
  val foo: Int
  def bar: Int = 10
  println("In A: foo: " + foo + ", bar: " + bar)
}

class B extends A {
  val foo: Int = 25
  println("In B: foo: " + foo + ", bar: " + bar)
}

class C extends B {
  override def bar: Int = 99
  println("In C: foo: " + foo + ", bar: " + bar)
}
```

```
scala> new C
In A: foo: 0, bar: 99
In B: foo: 25, bar: 99
In C: foo: 25, bar: 99
```

这里将 bar 定义成 def 之所以能解决问题，是因为 def 这个方法体不属于主构造器，因此不参与类初始化。此外，因为类 C 中重载了 bar，多态会特别选择使用这个重载的定义。因此，3 个 println 语句中的 bar 都会调用类 C 中的重载定义。

这种方法的一个缺点是每次调用都要评估。Scala 也遵从统一访问原则[①]，所以在超类中定义一个参数方法不会阻止在子类中将它重载为一个 val，这会导致令人迷惑的行为再次出现，从而破坏所有的精心规划。

lazy val

另外一种避免这种意外的方法是将 bar 声明为 lazy val。lazy val 在初次访问时初始化。而常规的 val，又叫静态变量，是在定义时初始化的。下面是使用了 lazy val 的程序：

```
trait A {
  val foo: Int
  lazy val bar = 10
  println("In A: foo: " + foo + ", bar: " + bar)
}

class B extends A {
  val foo: Int = 25
  println("In B: foo: " + foo + ", bar: " + bar)
}

class C extends B {
  override lazy val bar = 99
  println("In C: foo: " + foo + ", bar: " + bar)
}

new C()
```

这个程序的结果也符合预期：

```
In A: foo: 0, bar: 99
```

① Odersky, Spoon, Venners,《Scala 编程》，词汇在线。[Odeb]。

```
In B: foo: 25, bar: 99
In C: foo: 25, bar: 99
```

将 bar 声明为 lazy val 意味着在特质 A 构造期间就将它初始化成了 99，因为这是它首次被访问的地方。

lazy val 使用编译器生成的方式初始化，这里将调用特质 C 中 bar 的重载版本。注意，lazy val 的特点是将高成本的初始化过程尽可能推迟到最后时刻（有时可能永远也不进行初始化）。这不是本章的目的：在这个例子中，用 lazy val 是为了确保运行时正确的初始化顺序。

不过，要注意的是，lazy val 也有一些缺点：

1．由于在底层发生同步，这会引起轻微的性能成本；

2．不能声明抽象 lazy val；

3．使用 lazy val 容易产生循环引用，从而导致首次访问时发生栈溢出错误，甚至可能发生死锁；

4．如果在对象间做了声明而 lazy val 间的循环依赖却不存时，就可能会发生死锁，这种情况也许非常微妙，不易觉察[①]。

预初始化字段

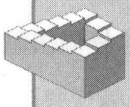
使用预初始化字段（也就是大家所知道的早期初始化器）也可以达到相同的效果：

```
trait A {
  val foo: Int
  val bar = 10
  println("In A: foo: " + foo + ", bar: " + bar)
}

class B extends A {
  val foo: Int = 25
  println("In B: foo: " + foo + ", bar: " + bar)
}


class C extends {
  override val bar = 99
```

① SIP-20，改进的 lazy val 初始化目标是通过重新实现初始化机制极大地减少死锁的可能性。

```
} with B {
  println("In C: foo: " + foo + ", bar: " + bar)
}

scala> new C
In A: foo: 0, bar: 99
In B: foo: 25, bar: 99
In C: foo: 25, bar: 99
```

这段程序与原来的程序的唯一差别，就是 bar 在类 C 的早期字段定义从句中初始化。早期字段定义从句紧跟着 extends 关键字[①]后的大括号，它是子类的一部分，在超类构造器之前运行[②]。这样就可以确保 bar 在特质 A 被构造之前即被初始化。



用什么方法解决潜在的初始化顺序问题因不同用户用例而不同。

如果每次访问评估表达式的成本不是太高，也许会用定义的方法。或者只要能避免循环依赖，就可以用 lazy val 的方法，这对用户的类来说也许是最简单的解决方案。或者，如果用户很清楚他们应该使用早期字段定义，那么简单地使用原来的抽象 val 也是一个不错的选择。

① Odersky，《Scala 语言规范》，5.1.6 节。[Ode14]

② 有关初始化顺序的更深入的讨论，参见第 3 章。

第 5 章

集合操作

Scala 有着丰富的集合库，提供了许多不同集合类型上的函数和操作。尽管如此，有时候你还是希望增加自己的“实用工具”操作符。

这是一个尝试对多个集合实例的大小求和的例子（次优的，后面还有更多例子）。例如，对 `Vector("a")`，`List("b", "c")` 和 `Array("d", "e", "f")` 几个集合的大小求和，结果应该是 1 + 2 + 3 = 6。

```
scala> sumSizes(Seq(Vector("a"), List("b", "c"),
         Array("d", "e", "f")))
res4: Int = 6
```

你是想让这个实用工具函数能操作集合数据类型以便这个函数能广泛应用，例如 `Iterable`，显然，这个函数设计的本意是：无论传入的参数具体是什么集合类型，都能有相同方式的操作行为。

让我们看看以下代码的执行结果会是什么。

```
def sumSizes(collections: Iterable[Iterable[_]]): Int =
  collections.map(_.size).sum

sumSizes(List(Set(1, 2), List(3, 4)))
sumSizes(Set(List(1, 2), Set(3, 4)))
```

可能的结果

1. 打印出：

```
Int = 4
Int = 4
```

2. 打印出：

```
Int = 4
Int = 2
```

3. 打印出：

```
Int = 2
Int = 4
```

4. 打印出：

```
Int = 2
Int = 2
```

解释

你可能会问 sumSizes 是否会以某种方式对集合计数而不是计算集合的大小呢？这会使两个语句像选项 4 那样都打印出 2。可是，如你所见的程序执行结果，正确的答案是 2：

```
scala> sumSizes(List(Set(1, 2), List(3, 4)))
res5: Int = 4

scala> sumSizes(Set(List(1, 2), Set(3, 4)))
res6: Int = 2
```

为了理解这道谜题，你需要回想一下 Scala 集合库的另外一个特征：操作符一般会保持输入的集合类型不变！

有 Java 背景的开发者可能认为转换 Iterable 会产生一个遵从 Iterable 接口的结果，可能是任何实现类型。Scala 集合则更进一步，返回一个与输入类型[①]一样的 Iterable。这意味着下面语句的结果也是一个 set 类型，因此不能有重复的实例。

```
Set(List(1, 2), Set(3, 4)).map(_.size)
```

结果就是，这个中间值只有一个元素：

```
scala> Set(List(1, 2), Set(3, 4)).map(_.size)
res7: scala.collection.immutable.Set[Int] = Set(2)
```

这个 set 的行为解释了观察到的结果。

---

① 参见 Scala 文档 CanBuildFrom。[EPF]

把这个题目加入谜题，是因为事实上 sumSize 类型声明里没有出现 Set 类型。也没有明显的警告标记能使函数的作者或调用者考虑 Set 的唯一约束的影响，或者的确还有其他 Iterable 子类型也会有类似引起非本意结果的其他约束。

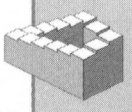
讨论

如何才能避免这种意外结果呢？在本例中一个可能的选择是将外集合转换成一种已知的类型，例如使用 toSeq：

```
def sumSizes(collections: Iterable[Iterable[_]]): Int =
  collections.toSeq.map(_.size).sum

scala> sumSizes(List(Set(1, 2), List(3, 4)))
res0: Int = 4

scala> sumSizes(Set(List(1, 2), Set(3, 4)))
res1: Int = 4
```

在这个特定的例子中，你甚至还可以做得更好。可以用 fold 来实现，就能避免这个问题并通过外部集合排除其中的一次 iterations：

```
def sumSizes(collections: Iterable[Iterable[_]]): Int =
  collections.foldLeft(0) {
    (sumOfSizes, collection) => sumOfSizes + collection.size
}

scala> sumSizes(List(Set(1, 2), List(3, 4)))
res8: Int = 4

scala> sumSizes(Set(List(1, 2), Set(3, 4)))
res9: Int = 4
```



要密切关注对集合进行操作的方法其输入类型是什么。如果你不是必需保留输入类型，也可以考虑用已知的特性构建自己的中间类型。

第 6 章

参数类型

Scala 简洁的语法和丰富的函数式原型结合，可以让你写出强大的优雅的程序。然而这些语句的复杂性使人很难理解其整体意图。为了便于理解，声明一个命名合适的帮助方法常常是个好主意。

例如，如果你需要多次用一个简单的“生成器”函数 f 的结果初始化 arg 值，可以用 fold（是递归计算的一种自然的选择）干净巧妙地实现。

```
// n applications of f to arg: f(f(...f(arg)))
val result = (1 to n).foldLeft(arg) { (acc, _) => f(acc) }
```

这对你的问题一定是个简洁的解决方案。但是，要弄明白这一行确切地做了什么，还需要仔细审视代码。

你可能试图将它抽取成一个帮助函数以使这行更容易理解。对于这个函数，我们先看看用单个参数列表的版本 applyNMulti，再看克里化的版本 applyNCurried。然后就可以将这些帮助函数应用到两个生成器函数中：简单的 nextInt 版本只应用于 Int 类型，还有一个更通用的版本 nextNumber 可以用 typeclass 模式[①]处理所有的数字类型。

让我们看看在 REPL 命令行里执行以下代码会是什么执行结果。

```
def applyNMulti[T](n: Int)(arg: T, f: T => T) =
  (1 to n).foldLeft(arg) { (acc, _) => f(acc) }

def applyNCurried[T](n: Int)(arg: T)(f: T => T) =
  (1 to n).foldLeft(arg) { (acc, _) => f(acc) }
def nextInt(n: Int) = n * n + 1

def nextNumber[N](n: N)(implicit numericOps: Numeric[N]) =
  numericOps.plus(numericOps.times(n, n), numericOps.one)
```

① Sobral，“隐式技巧——类型类模式”。[Sob10]

```
println(applyNMulti(3)(2, nextInt))
println(applyNCurried(3)(2)(nextInt))
println(applyNMulti(3)(2.0, nextNumber))
println(applyNCurried(3)(2.0)(nextNumber))
```

可能的结果

1. 第 1、2、4 个语句打印出：

```
677
677
677.0
```

第 3 个语句编译失败。

2. 第 1、2 个语句打印出：

```
677
677
```

第 3、4 个语句编译失败。

3. 第 1、3、4 个语句打印出：

```
677
677
677.0
```

第 4 个语句编译失败。

4. 打印出：

```
677
677
677.0
677.0
```

解释

用简单的 `nextInt` 生成器函数可以成功运行，但通用版本的例子为什么会出错呢？`Double` 所需要的数字操作符也在范围内啊，你可以用下面的代码验证：

```
scala> val doubleOps = implicitly[Numeric[Double]]
doubleOps: Numeric[Double] =
  scala.math.Numeric$DoubleIsFractional$@7c54c0ca
```

也许编译器只是不喜欢我们的通用版本？你确定语句 3 和语句 4 都会失败吗？

如所发生的那样，并非如此。只有非克里化版本应用通用函数 nextNumber 让编译器发生了错误，所以正确的答案是 1：

```
scala> println(applyNMulti(3)(2, nextInt))
677

scala> println(applyNCurried(3)(2)(nextInt))
677

scala> println(applyNMulti(3)(2.0, nextNumber))
<console>:10: error: could not find implicit value for
  parameter numericOps: Numeric[N]
                println(applyNMulti(3)(2.0, nextNumber))
                                           ^


scala> println(applyNCurried(3)(2.0)(nextNumber))
677.0
```

编译器报了什么错呢？错误消息说明编译器并不是像你预期的那样在寻找 Numeric[Double]类型，而是在寻找 Numeric[N]..？

N 不是用 2.0 作为第一个参数已经绑定成 Double 了吗？

是的，它是已经绑定成 Double 了。但关键是这对编译器并没有帮助。编译器试图分别满足参数列表中的每个参数的类型需求，因此不能用同一个参数列表中其他参数类型的信息。在这个例子中，事实上，通用类型 *N* 被绑定到特定的类型 Double，当编译器搜索合适的 numericOps 时，它还不可用。即便这个绑定的 *N* 参数在参数列表中出现在 nextNumber 之前也一样。

克里化的例子则与之相反，*N* 作为前面的参数列表评估的一部分已经被绑定到 Double 了，与相同列表中的早参数相对。nextNumber 应用于其中的偏函数确实期望一个 Double 参数，正如它的类型标识符所示：

```
scala> applyNCurried(3)(2.0) _
res9: (Double => Double) => Double = <function1>
```

因为编译器知道，处理 nextNumber 时就会将 *N* 绑定到 Double，这样它就能找到范围内合适的隐式的 Numeric[Double]值，从而使这个变量得以成功执行。

顺便说一下，这种“类型信息蔓延”也就是为什么这么多的 Scala 标准函数会有克里化定义的原因之一。

讨论

你可以通过显式指定类型使未克里化的变量成功运行：

```
scala> println(applyNMulti(3)(2.0, nextNumber[Double]))
677.0

scala> println(applyNMulti[Double](3)(2.0, nextNumber))
677.0
```

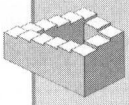
尽管感觉有些不必要，因为参数 2.0 已经携带了类型信息。如果初始值作为变量而不是常量传递，这也不是一个符合实际的解决方案，因为改变变量的类型（从 Double 变成 Float）会引起本应该能避免的编译错误。

```
val firstVal = 2.0

scala> println(applyNMulti(3)(firstVal, nextNumber[Double]))
677.0

// refactored to use a Float
val firstVal = 2.0f
scala> println(applyNMulti(3)(firstVal, nextNumber[Double]))
<console>:11: error: type mismatch;
  found : Double => Double
  required: AnyVal => AnyVal
    println(applyNMulti(3)(firstVal, nextNumber[Double]))
                                                  ^
```

也要注意，涉及 Numeric 的 typeclass 模式不是这个问题的根本：任何类型约束，例如类型绑定，都能引起同样的行为，如前面所述，克里化版本却可以得到期望的结果：

```
def nextAnyVal[N <: AnyVal](n: N) = n

scala> applyNCurried(3)(2.0)(nextAnyVal)
res21: Double = 2.0

scala> applyNMulti(3)(2.0, nextAnyVal)
<console>:10: error: type mismatch;
  found : Nothing => Nothing
```

```
required: Double => Double
            applyNMulti(3)(2.0, nextAnyVal)
                                ^
```



一个类型参数的信息只对克里化调用中随后的参数列表是可用的，对同一列表中的其他参数是不可用的。当定义一个有参数的方法时，参数的类型约束只有相同方法的早参数绑定的类型是已知时才能满足，这时要用克里化的方法来定义而不是单个参数列表的方法。

第 7 章

闭包

函数值，尤其是匿名函数，提供一种方便和简洁的方式来创建和传递“便携的”代码片段。定义函数时，通过允许函数引用范围内的值从而使函数功能得以增强，而不是只能引用直接的函数参数。

以下代码为一个值的集合创建了“延迟访问器”，并可以稍后调用。

在 REP 命令行中执行这个代码的结果会是什么呢？

```
import collection.mutable.Buffer

val accessors1 = Buffer.empty[() => Int]
val accessors2 = Buffer.empty[() => Int]

val data = Seq(100, 110, 120)
var j = 0
for (i <-0 until data.length) {
  accessors1 += (() => data(i))
  accessors2 += (() => data(j))
  j += 1
}

accessors1.foreach(a1 => println(a1()))
accessors2.foreach(a2 => println(a2()))
```

可能的结果

1. 第 1 个语句打印出：

```
100
110
120
```

第 2 个语句抛出异常 IndexOutOfBoundsException。

2. 两个语句都打印出：

```
100
110
120
```

3. 两个语句都编译失败，报错错误消息："not found: value data."

4. 两个语句都打印出：

```
120
120
120
```

解释

因为当函数被调用时，data，i 和 j 都已不在范围内了，你可能怀疑这些代码能编译吗？或者你可能好奇是否那些函数都能看到 data(i)和 data(j)的最终值，你期望的结果是两个都打印出：

```
120
120
120
```

正如所发生的那样，代码确实编译了，第 1 个语句打印出期望的值：100,110,120。第 2 个语句不执行，立即抛出异常：

```
scala> accessors1.foreach(a1 => println(a1()))
100
110
120

scala> accessors2.foreach(a2 => println(a2()))
java.lang.IndexOutOfBoundsException: 3
  at scala.collection.LinearSeqOptimized$class.apply(
    LinearSeqOptimized.scala:51)
  at scala.collection.immutable.List.apply(List.scala:85)
  at $anonfun$1$$anonfun$apply$mcVI$sp$2.apply$mcI$sp(
    <console>:16)
  at $anonfun$1.apply(<console>:10)
  ...
```

因此，正确的答案是 1。

在仔细检查到底是 i 和 j 之间的哪点不同导致了观察到的行为之前，让我们看看 Scala 是如何允许函数体访问这些变量的会很有帮助。

Scala 允许函数体引用没有显式函数参数但函数被构造时又在范围内的变量。当调用这些函数时，为了在不同的范围内访问这些自由变量，Scala 将它们封闭成一个闭包。

封闭一个自由变量并非是使用时取这个变量值的一个快照，而是将引用捕捉变量的一个字段加到函数对象中。对这个例子克里化，捕获的 val 简单地由值表示，而捕获一个 var 的结果是引用了 var 自己。

举个例子，考虑一下这个 fun 方法：

```
def fun: () => Int = {
  val i = 1
  var j = 2
  () => i + j
}
```

从 fun 方法返回的函数是一个闭包，捕获一个 val 变量 i 和一个 var 变量 j。你可以用-print 选项调用 scala，它会打印出删除了所有 scala 特定特征的代码，从而可以检查编译器是如何对 i 和 j 区别对待的。

```
def fun(): Function0 = {
  val i: Int = 1;
  var j: runtime.IntRef = new runtime.IntRef(2);
  {
    (new anonymous class anonfun$fun$1(Illustration.this,
        i, j): Function0);
  }
};
```

看到差异了吗？val 存储为常规的 Int；而 var 变成一个 scala.runtime.IntRef，一个对可变（Java 的 mutable）int 的引用。

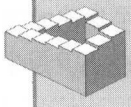
这里对观察到的行为的解释还是直观的：当每次创建 accessors1 函数时，它就会捕获当前的 i 值，所以每次调用时都会打印出期望的结果。而 accessors2 函数，每次捕获一个含有 j 值的可变的 IntRef 对象的引用，这个值随时间而变化。

第一次调用 accessors2 函数时，j 的值已经是 3 了。因为数据只有 3 个元素，所以调用 data(j) 就触发了 IndexOutOfBoundsException 例外。

讨论

阻止这种问题发生的最鲁棒的方式是避免 var，这也是较好的 Scala 样式。如果不能避免使用 var，但仍然希望用闭包在创建时捕获它的值，可以将它的值分配给一个临时 val 来“冻结”var。请看下面这个例子：

```
import collection.mutable.Buffer

val accessors2 = Buffer.empty[() => Int]

val data = Seq(100, 110, 120)
var j = 0
for (i <- 0 until data.length){
  val currentJ = j
  accessors2 += (() => data(currentJ))
  j += 1
}

scala> accessors2.foreach(a2 => println(a2()))
100
110
120
```



要避免在闭包里捕获引用任何可变 var 或可变对象的自由变量。如果你需要封闭任何可变的值，可以抽取一个不变值赋给 val，然后在函数中使用这个 val。

第 8 章

Map 表达式

Scala 的 for 表达式为调用以 map 和 flatmap 为基础的强大函数式构造提供了优雅的语法。for 表达式在 Scala 中如此广泛使用，以至于理解如何将 for 表达式提糖[①]成 map、flatMap、withFilter 和 foreach 调用，是学习这门语言时常用的练习。这对调试程序尤其有用，因为有时候仅仅是出于调试的目的，你可能倾向于手工提糖一个 for 表达式。

在这个谜题中，我们将一个使用了模式匹配的 for 表达式与由转换得到的相同的 for 表达式的提糖版本进行了比较：

```
for (i <- expr) yield fun(i)
```

与下面的代码比较：

```
expr map { i => fun(i) }
```

以下代码在 REPL 中执行会是什么结果呢？

```
val xs = Seq(Seq("a", "b", "c"), Seq("d", "e", "f"),
             Seq("g", "h"), Seq("i", "j", "k"))
val ys = for (Seq(x, y, z) <-xs) yield x + y + z
val zs = xs map { case Seq(x, y, z) => x + y + z }
```

可能的结果

1. 评估 ys 和 zs 抛出 MatchError 异常。
2. ys 和 zs 都评估为：

---

① 在计算机科学中，在编程语言中使用语法糖（Syntactic Sugar）的语法以便于代码更可读、易于理解和表达。这种设计使代码更“甜”，即表达更清晰，是人们比较偏好的样式。提糖是对加了语法糖的代码脱糖，提出代码中的糖分，还原为代码的本质。——译者注

```
Seq(abc,def,ijk)
```

3. 评估 ys 抛出 MatchError 异常，zs 评估为：

```
Seq(abc,def,ijk)
```

4. ys 评估为：

```
Seq(abc,def,ijk)
```

评估 zs 抛出 MatchError 异常。

解释

仔细查看这段代码，你会注意到 xs 中的 Seq("g", "h") 与 for 表达式中及提糖后的 map 中所用的 Seq(x, y, z) 模式并不匹配。所以你在想 ys 和 zs 究竟能不能透明地跳过这个值呢？如果不能，两个语句都会抛出 MatchError 异常。但是，for 表达式和提糖后的 map 调用一定会有相同的表现吗？

你的推断是错误的。正确的答案是 4：

```
scala> val ys = for (Seq(x, y, z) <-xs)yield x + y + z
ys: Seq[String] = List(abc, def, ijk)

scala> val zs = xs map { case Seq(x, y, z) => x + y + z }
scala.MatchError: List(g, h) (of class
    scala.collection.immutable.$colon$colon)
  at $anonfun$1.apply(<console>:8)
  at $anonfun$1.apply(<console>:8)
  at scala.collection.TraversableLike$$anonfun$map$1.
    apply(TraversableLike.scala:245)
  ...
```

for 表达式跳过了有问题的值，而直接用 map 的调用却失败了。为了解释为什么会是这样，让我们先看看下面这个简单的 for 表达式：

```
for (i <-0 to 1) yield i + 1
```

这是用与上例同样的模式提糖后的结果：

```
0 to 1 map { i => i + 1 }
```

让我们用-Xprint:parser[①]选项启动 REPL 来看看提糖的过程：

```
scala> for (i <-0 to 1) yield i + 1
```

① 编译器参数-Xprint:<phase>在特定的编译期阶段之后打印程序代码。Scala - Xshowphases 显示编译阶段的列表，第一个阶段是解析器。

```
[[syntax trees at end of        parser]] // <console>
    ...
    val res2 = 0.to(1).map(((i) => i.$plus(1)))
```

在 Scala 中，i <-0 to 1 的语法叫生成器。使用对一个简单变量赋值的生成器，很容易让人忘记生成器左边不是一个简单变量，而是一个模式，如代码例子中的 Seq(x, y, z) <-xs 所示。Scala 编译器用各种非凡模式（如对多个简单变量赋值的模式）对生成器提糖。请看下面这个表达式：

```
for (pattern <-expr)yield fun
```

这个代码可以重写成：

```
expr map { case pattern => fun }
```

例如：

```
scala> for (i@j <- 0 to 1) yield i + j
[[syntax trees at end of        parser]] // <console>
    ...
    val res0 = 0.to(1).map(((x$1) => x$1:
          @scala.unchecked match {
      case (i @ (j @ _)) => i.$plus(j)
    }))
```

到目前为止，这与该例中我们自己的提糖还是一致。但语言规范也规定非凡模式是 withFilter 调用的附加[①]。因此下面的表达式：

```
for (pattern <- expr) yield fun
```

实际上就变成：

```
expr withFilter {
  case pattern => true
  case _ => false
} map { case pattern => fun }
```

是 withFilter 调用透明地“剔除”了不匹配的值，而在我们的提糖过程中正是这些不匹配的值引起了 MatchError 异常。

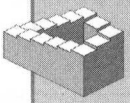
讨论

你可以将 for 表达式中的模式匹配生成器当成含有“if matches”的防卫（防

① filter 是创建一个新的集合并因而产生对原集合全部运行的负荷。与 filter 不同，withFilter 是一个简单的“视图”，它以 lazily 的方式限制数据项传递到后续 map，flatMap，foreach 和 withFilter 调用，withFilter 是特别为高效地链接这些操作而设计的。

卫被提糖成 withFilter 调用)。将它加到提糖后的版本中就能产生期望的结果了:

```
scala> xs withFilter { case Seq(x, y, z) => true; case _ =>
     |   false } map { case Seq(x, y, z) => x + y + z }
res1: Seq[String] = List(abc, def, ijk)
```



因为 for 表达式在 Scala 代码中非常普遍，花些时间让自己熟悉 for 表达式是如何提糖的还是很值得的。

第 9 章

循环引用变量

随着程序逐渐变得越来越庞大，你可能会用到有循环依赖的模块。但可靠地初始化这些模块可能是个挑战。

在 REPL 中执行以下代码会是什么结果呢？

```
object XY {
  object X {
    val value: Int = Y.value + 1
  }
  object Y {
    val value: Int = X.value + 1
  }
}

println(if (math.random > 0.5) XY.X.value else XY.Y.value)
```

可能的结果

1. 打印出：

```
1
```

2. 打印出：

```
2
```

3. 打印出：

```
1
```

或者打印出：

```
2
```

4. 抛出一个运行时例外。

解释

你也许会问 Scala 编译器是否能处理这种循环定义呢？或者在运行时是否会进入无限循环呢？

即便 Scala 真能处理这种定义而不产生冲突，编译器是否会按声明的顺序初始化值呢。本例子中程序随机地打印出先声明的值(XY.X.value)，然后打印出后声明的值(XY.Y.value)的任何值，所以你期望看到的是个不确定的结果。

另外一种可能，你可能猜测，第一次访问正在初始化的对象时看到的是缺省值，因为这时对象还未初始化，所以结果就是每次都打印出 1。

事实上，正确的答案是 2。每次都打印出 2：

```
scala> println(if (math.random > 0.5) XY.X.value else
         XY.Y.value)
2

scala> println(s"X: ${XY.X.value} Y: ${XY.Y.value}")
X: 2 Y: 1
```

几个:reset 命令之后，你应该最终看到如下结果[①]：

```
scala> println(if (math.random > 0.5) XY.X.value else
         XY.Y.value)
2

scala> println(s"X: ${XY.X.value} Y: ${XY.Y.value}")
X: 1 Y: 2
```

为了理解究竟发生了什么，让我们先证明 Scala 编译器处理 val 定义中的循环是没有问题的。不过它确实需要至少一次显式的类型说明：

```
scala> lazy val x = y; lazy val y = x
<console>:12: error: recursive value y needs type
       lazy val x = y; lazy val y = x
                       ^


scala> lazy val x: Int = y; lazy val y = x
```

① :reset 命令告诉 REPL“忘记”所有的定义，这可以让你重新初始化 XY.X.value 和 XY.Y.value。

```
x: Int = <lazy>
y: Int = <lazy>
```

语言规范上说“对象定义的值按 lazily[①]方式实例化”，并进一步解释说：一个对象的确是被看作“大约等于[. . .]一个 lazy 值”。这也解释了为什么随机选择打印 XY.X.value 或 XY.Y.value 不会影响输出结果：与声明顺序无关，因为声明对象时还没有初始化它的值，而是在访问时才初始化。要打印的对象总是先初始化，先初始化的对象值都是 2。

但是先初始化的对象的值怎么会是 2 呢，你访问的时候为什么没有进入无限循环呢？让我们用 scalac -print 命令对 XY 对象进行编译可以看到编译器输出结果如下：

```
[[syntax trees at end of     cleanup]] // XY.scala
package <empty> {
  object XY extends Object {
    def <init>(): XY.type = {
      XY.super.<init>();
      ()
    }
  };
  object XY$X extends Object {
    private[this] val value: Int = _;
    <stable> <accessor> def value(): Int = XY$X.this.value;
    def <init>(): XY$X.type = {
      XY$X.super.<init>();
      XY$X.this.value = XY$Y.value().+(1);
      ()
    }
  };
  object XY$Y extends Object {
    private[this] val value: Int = _;
    <stable> <accessor> def value(): Int = XY$Y.this.value;
    def <init>(): XY$Y.type = {
      XY$Y.super.<init>();
      XY$Y.this.value = XY$X.value().+(1);
      ()
    }
  }
}
```

当你随机地访问选择的对象时发生了什么呢？假设你正在试图获取 XY.Y.value：

---

① Odersky，《Scala 语言规范》，5.4 节。[Ode14]

1. XY 通过调用它的构造器来初始化。这没什么特别的。

2. 而 XY$Y 通过调用它的构造器初始化，这个构造器又试图通过访问器 XY$X.value() 获得 X 的值。

3. 调用 XY$X.value() 触发了 XY$X 通过其构造器再次被初始化。因此，它现在试图调用 XY$Y.value() 来检索 Y 值。

4. 此时，Y 还没有初始化，所以你似乎就在无限循环的边缘。但现在“神奇的事”发生了：JVM 规范明确规定实例不能被多次初始化[①]。结果 XY$X 直接调用 XY$Y 的访问器方法 value()，而由于这个值此时还没有定义，所以调用结果就返回 Int 类型的缺省值 0。

5. 假设这个值是 0，XY$X 的构造器现在就能完成初始化 XY$X.this.value，将它初始化为 1 并返回。

6. 最后，XY$Y 的构造器成功调用 XY$X.value() 并返回值 1。

7. 对于给定的值 1，XY$Y 构造器完成对 XY$Y.this.value 赋值，将它的值设为 2。

如果此时你恰好选择打印 XY.X.value 值，此时就将它初始化成这个保留的角色。这就解释了为什么首次访问的对象总是赋值为 2，另一个对象赋值为 1。

讨论

当你将这个观察到的结果与类似的循环定义所产生的结果进行比较时，就会觉得这个结果更令人惊讶了。例如，Scala 语言规范中说对象是“大约等于[lazy values]”[②]，你可以试试下面的代码：

```
object XY2 {
  lazy val xvalue: Int = yvalue + 1
  lazy val yvalue: Int = xvalue + 1
}

scala> println(if (math.random > 0.5) XY2.xvalue else
         XY2.yvalue)
java.lang.StackOverflowError
```

① Lindholm 等，《Java 虚拟机规范》，5.5 节。[Lin13]。

② 关于变初始化选项更详细的讨论，参见第 4 章。

```
  ...
  at XY2$.xvalue(<console>:8)
  at XY2$.yvalue$lzycompute(<console>:9)
  at XY2$.yvalue(<console>:9)
  at XY2$.xvalue$lzycompute(<console>:8)
  at XY2$.xvalue(<console>:8)
```

或者你可以继续坚持用对象，但要将对象放进一个封闭的类里而不是对象中：

```
class XY3 {
  object X {
    val value: Int = Y.value + 1
  }
  object Y {
    val value: Int = X.value + 1
  }
}

scala> val xy3 = new XY3()
xy3: XY3 = XY3@770b07b9

scala> println(if (math.random > 0.5) xy3.X.value else
         xy3.Y.value)
java.lang.StackOverflowError
  ...
  at XY3.Y$lzycompute(<console>:11)
  at XY3.Y(<console>:11)
  at XY3$X$.<init>(<console>:9)
  at XY3.X$lzycompute(<console>:8)
  at XY3.X(<console>:8)
  at XY3$Y$.<init>(<console>:12)
  at XY3.Y$lzycompute(<console>:11)
```

这两种情况都漏掉了由 JVM 所规定的“不能对同一个实例初始化多次”规则所提供的“无限循环保护”。编译器允许两个函数相互调用，但会在运行时抛出异常。

在第二个例子中，`Y$lzycompute` 创建 `Y` 的一个新实例并分配给 `XY.Y` 单元。这会尝试访问 `XY.X` 从而触发 `X$lzycompute`，而且因为此时 `XY.Y` 还没有被初始化，所以会再次调用 `Y$lzycompute`。`Y$lzycompute` 就会试图创建另外一个 `Y` 的实例，如此循环。

此外，还有一个选择，就是你还可以少用些 lazy：

```
object XY4 {
  lazy val xvalue: Int = yvalue + 1
  val yvalue: Int = xvalue + 1
}

scala> println(if (math.random > 0.5) XY4.xvalue else
         XY4.yvalue)
2

scala> println(s"X: ${XY4.xvalue} Y: ${XY4.yvalue}")
X: 1 Y: 2
```

现在，不再是访问顺序决定它们的值了：XY4 初始化之后会立即评估 yvalue。这又触发了 xvalue 的评估，从而能看到 yvalue 的缺省值从 0 变成了 1，yvalue 值变成了 2。按照什么顺序声明 xvalue 和 yvalue 还是没有影响，尽管：

```
object XY4a {
  val yvalue: Int = xvalue + 1
  lazy val xvalue: Int = yvalue + 1
}

scala> println(if (math.random > 0.5) XY4a.xvalue else
         XY4a.yvalue)
1

scala> println(s"X: ${XY4a.xvalue} Y: ${XY4a.yvalue}")
X: 1 Y: 2
```

你也可以完全不用 lazy 值：

```
object XY5 {
  val xvalue: Int = yvalue + 1
  val yvalue: Int = xvalue + 1
}

scala> println(if (math.random > 0.5) XY5.xvalue else
         XY5.yvalue)
1

scala> println(s"X: ${XY5.xvalue} Y: ${XY5.yvalue}")
X: 1 Y: 2
```

这里，在初始化 XY5 时，xvalue 和 yvalue 都立即被评估。xvalue 试

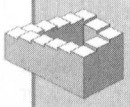
图检索还未赋值的 yvalue 的值，再次看到缺省值是 0，接着被设置成 1，然后 yvalue 一直是 2。这个问题是可预测到的，以至于 XY5 一被定义编译器就发出了警告。

```
scala> object XY5 {
         val xvalue: Int = yvalue + 1
         val yvalue: Int = xvalue + 1
       }
<console>:8: warning: Reference to uninitialized value yvalue
         val xvalue: Int = yvalue + 1
                           ^

defined object XY5
```

此外，与其他例子不同的是，这里声明的顺序决定了 xvalue 和 yvalue 的值。调用顺序不同，值也不同。

```
object XY5a {
  val yvalue: Int = xvalue + 1
  val xvalue: Int = yvalue + 1
}

scala> println(s"X: ${XY5a.xvalue} Y: ${XY5a.yvalue}")
X: 2 Y: 1
```

总之，循环依赖和定义是很诡异的，且难以知道其原因。某些方式依赖声明的顺序，另一些方式则依赖初始化的顺序，还有一些会导致无限循环。在尽可能的地方都要避免使用这种方法。



在尽可能的地方都要避免循环依赖和定义。如果你实在无法找到替代的方法来删除循环，那也要确保你理解了所有组件和值的初始化行为。一定要进行充分的测试以确保代码结果是你期望的，如果元素初始化的顺序是不确定时尤其要注意。

第 10 章

等式的例子

Scala 的 case class 可以简单地用工厂方法来表示实体、抽取器或几个不用 for 的便利方法：

```
class Country(val isoCode: String, val name: String)
case class CountryCC(isoCode: String, name: String)

val homeOfScala = new Country("CH", "Switzerland")
val homeOfScalaCC =
  CountryCC("CH", "Switzerland") // factory method

scala> println(homeOfScala equals
         new Country("CH", "Switzerland"))
false

scala> println(homeOfScalaCC equals
         CountryCC("CH", "Switzerland"))
true

scala> println(homeOfScala.toString)
$line348.$read$$iw$$iw$Country@39eb8ede

scala> println(homeOfScalaCC.toString)
CountryCC(CH,Switzerland)
```

为了让你对这个谜题有更好的理解，我们将跟踪 hashCode 调用，这是其中的一个便利方法。我们给 case class 的声明或实例化混进“调试”特质，然后将 case class 实例增加到 HashSets，并演示如何使用 hashCode。

我们先看看在 REPL 中执行下面的代码会发生什么吧！

```
trait TraceHashCode {
  override def hashCode: Int = {
    println(s"TRACE: In hashCode for ${this}")
```

```
    super.hashCode
  }
}


// mix in trait at instantiation
case class Country(isoCode: String)
def newSwitzInst = new Country("CH") with TraceHashCode


// mix in trait at declaration time
case class CountryWithTrace(isoCode: String) extends
  TraceHashCode
def newSwitzDecl = CountryWithTrace("CH")


import collection.immutable.HashSet
val countriesInst = HashSet(newSwitzInst)
println(countriesInst.iterator contains newSwitzInst)
println(countriesInst contains newSwitzInst)


val countriesDecl = HashSet(newSwitzDecl)
println(countriesDecl.iterator contains newSwitzDecl)
println(countriesDecl contains newSwitzDecl)
```

可能的结果

1. 打印出：

```
true
TRACE: In hashCode for Country(CH)
true
true
TRACE: In hashCode for CountryWithTrace(CH)
true
```

2. 打印出：

```
true
TRACE: In hashCode for Country(CH)
true
true
TRACE: In hashCode for CountryWithTrace(CH)
false
```

3. 打印出：

```
true
TRACE: In hashCode for Country(CH)
false
true
TRACE: In hashCode for CountryWithTrace(CH)
false
```

4. 打印出：

```
true
TRACE: In hashCode for Country(CH)
true
false
TRACE: In hashCode for CountryWithTrace(CH)
false
```

解释

为 case class 生成的 equal 和 hashCode 实现是基于结构等式的：如果两个实例有相同的类型和相等的构造器参数，那么这两个实例就应该是相等的。因为混进 TraceHashCode 特质不会影响结构，你可能会认为既然由 newSwitzInst 创建的实例是相等的且有相同的 Hash code，那么对 newSwitzDecl 应该也一样啊。如果这是对的，那么，countriesInst 就应该包含 newSwitzInst，countriesDecl 就应该包含 newSwitzDecl。

或者，你也许会问在声明时混进 TraceHashCode 特质是否会“封闭”为 CountryWithTrace 生成的结构等式呢。由 newSwitzDecl 创建的不同实例应该有不同的 hashcode 且它们是不相等的，因此由 newSwitzDecl 创建的第二个实例就不会是 countriesDecl 的一个成员。那么，你能确定无论是检查 set 还是 iterator 都没有什么区别吗？实际上，的确如此。在实例中混进 TraceHashCode 特质并不会影响 equal 和 hashCode 行为。但是将 CountryWithTrace 声明为从 TraceHashCode 继承就会封闭为 case class 生成的 hashCode 方法，所以由 newSwitzDecl 创建的新实例并没有出现在 set 中。Iterator 所依赖的 equal 实现也没有受到影响。

正确的答案是 2：

```
scala> println(countriesInst.iterator contains
         newSwitzInst)
true

scala> println(countriesInst contains newSwitzInst)
TRACE: In hashCode for Country(CH)
true

scala> println(countriesDecl.iterator contains
         newSwitzDecl)
true

scala> println(countriesDecl contains newSwitzDecl)
TRACE: In hashCode for CountryWithTrace(CH)
false
```

这是一个特殊的问题，因为你正在无意中违反了 equals/hashCode 协议，协议中说“如果两个对象相等[...]，则必须有相同的 hashcode”[①]，由 new SwitzInst 创建的两个实例被认为是相等的（有相同的 hashcode），所以在实例化时混进 TraceHashCode 特质并没有任何出乎意料的结果。

语言规范对 case class[②]的解释可以帮助我们弄明白发生了什么（我们的着重点）：

如果 case class 里没有定义这个方法，或者在不同于 AnyRef 的 case class 的某些基本类中也没有这个方法的具体定义，那么每个 case class 都会隐式地重载 scala.AnyRef 类的方法定义。

所以，只有当 case class 中没有方法的显式实现并且也没有从父类、父特质中继承方法的实现时，编译器才会生成重载。此外，这个重载方法生成的条件（在此例中是 equals 和 hashCode）是彼此相互独立的，所以 equals 和 hashCode 之间的一致性留给开发者来处理。

在这个例子中，编译器为 CountryWithTrace 的 equals 方法生成一个重载实现，所以比较由 newSwitzDecl 所创建的两个实例，newSwitzDecl == newSwitzDecl，评估结果为 true。

然而，hashCode 方法并没有被重载，所以在 TraceHashCode 里调用 super.hashCode 就调用了 AnyRef 的缺省实现，这与参考等式一致。因此，newSwitzDecl.hashCode ==newSwitzDecl.hashCode 返回 false，所以在 countriesDecl 集合中并没有发现由 newSwitzDecl 创建的新实例。

---

① 参见 Scaladoc 中的 scala.Any。[EPF]

② Odersky，《Scala 语言规范》，5.3.2 节。[Ode14]

至于 new Country("CH") with TraceHashCode 这个例子，当声明 case class Country 的时候，生成的重载就被编译器加进来，此时 equals 和 hashCode 都还没有显式地实现。newSwitzInst 创建新实例期间混入了 TraceHashCode 特质，此时 Country 已经有了一个基于结构等式的 equals 方法。从而 TraceHashCode 中的 super.hashCode 就会调用 Country 中的编译器生成的 hashCode 方法，这正符合我们的本意。

讨论

在实例化时增加“调试”特质似乎是一个办法。不过，如果不希望每次创建实例的时候都得混进 TraceHashCode 特质，还可以通过（临时地）创建一个 Country 子类来做到。

```
case class _Country(isoCode: String) // renamed
// use :paste in the REPL
class Country(isoCode: String) extends
  _Country(isoCode: String) with TraceHashCode
object Country {
  def apply(isoCode: String): Country = new Country(isoCode)
}
// ctrl-D to end :paste mode
def newSwitzSubcl = Country("CH")

scala> println(newSwitzSubcl == newSwitzSubcl)
true

scala> println(newSwitzSubcl.hashCode
         == newSwitzSubcl.hashCode)
TRACE: In hashCode for _Country(CH)
TRACE: In hashCode for _Country(CH)
true
```

继承 case class 不是一个好的做法。不过，通过“替换” case class 的工厂方法实现要好些。如果你定义了自己的方法，而编译器仍然试图生成 apply 方法，这会引起编译器错误。如果你希望在 case class 的伴生对象中重新定义标准的 apply 工厂方法，需要将 case class 声明为 abstract：

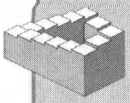
```
// use :paste in the REPL
abstract case class Country(isoCode: String)
```

```
object Country {
  def apply(isoCode: String): Country =
    new Country(isoCode) with TraceHashCode
}
// ctrl-D to end :paste mode
def newSwitzFact = Country("CH")

scala> println(newSwitzFact == newSwitzFact)
true

scala> println(newSwitzFact.hashCode
         == newSwitzFact.hashCode)
TRACE: In hashCode for Country(CH)
TRACE: In hashCode for Country(CH)
True
```

为方便起见，编译器还是会给伴生对象增加一个 unapply 的实现，所以你的 case class 仍然可以用模式匹配。但不能用 new 来创建一个实例（例如，new Country("CH")，因为 Country 是抽象类。

如果你对 case class 的声明存在疑惑，最简单的方法是避免用 super.hashCode，简单地确保 hashCode 的实现与结构等式一致。调用 isoCode.hashCode 可以满足这个需求，但是必须注意，isoCode 一定是 null。Scala 的 null 安全版本的 hashCode 方法##可以避免这个问题：

```
case class CountryWithTrace(isoCode: String) {
  // avoiding super.hashCode
  override def hashCode: Int = {
    println(s"TRACE: In hashCode for ${this}")
    isoCode.##
  }
}
def newSwitzHCImpl = CountryWithTrace("CH")

scala> println(newSwitzHCImpl == newSwitzHCImpl)
true

scala> println(newSwitzHCImpl.hashCode
         == newSwitzHCImpl.hashCode)
TRACE: In hashCode for CountryWithTrace(CH)
TRACE: In hashCode for CountryWithTrace(CH)
true
```



为 case class 提供你自己的 equal 或 hashCode 的实现时：

1. 如果只定义两个方法之一，要确保它遵从结构化等式。

2. 否则，两个方法都要根据 equals/hashCode 协议来实现。

第 11 章

lazy val

缺省情况下，Scala 会执行严格的评估：表达式在定义之后会立即评估。然而，如谜题 4 中所详细讨论的那样，Scala 也支持 lazy（非严格）评估，即变量直到第一次被引用时才会初始化。

严格或不严格的方式评估表达式，都有可能产生异常，这会引起一些有趣的行为。下面的程序例子就是这样一个情形。让我们看看它做了什么吧。

```
var x = 0
lazy val y = 1 / x

try {
  println(y)
} catch {
  case _: Exception =>
    x = 1
    println(y)
}
```

可能的结果

1. 抛出 ArithmeticException:/by zero 异常。

2. 打印出：

```
1
```

3. 打印出：

```
Infinity
```

4. 打印出：

```
<lazy>
```

解释

首先，让我们再仔细看看程序并思考一些可能的结果。y 值被声明为 lazy，所以直到首次访问时才对它初始化，也就是第一次调用 println 的时候。此时，y 还是未定义的并被标记为 lazy，程序可能会抛出一个运行时异常（被 0 除）：

```
scala> var x = 0
x: Int = 0

scala> lazy val y = 1 / x
y: Int = <lazy>
```

在 catch 程序块中，第二次调用 println 会输出当前仍未初始化的 y:<lazy>。此时重新给变量 x 赋值 1 也不会影响 y 的值。

或者，当进入 try 程序块时，y 还没被初始化，在 catch 块中第二次调用 println 可能试图再次初始化这个变量。结果就是再次抛出原来的“被 0 除”异常。

要不然就是 1 除以 0 结果肯定是算术里的无穷大（第 3 个候选答案）。

让我们在 REPL 中运行这个代码看看实际结果是什么吧：

```
scala> try {
         println(y)
       } catch {
         case _: Exception =>
           x = 1
           println(y)
         }
1
x: Int = 1
y: Int = <lazy>
```

很意外吧？正确的答案是 2！结果打印出 1，也许是 x=1 语句导致了这个结果？事实上，的确如此。除了延迟评估之外，lazy 值还有个有趣的特性：如果在初始化时抛出了异常，再次访问的时候就会重新计算[①]。这正是第二次调用 println 时所发生的，也就是为什么打印出 1。

---

① Odersky，《Scala 语言规范》，5.2 节。[Ode14]

讨论

因为将 y 定义为 lazy val，表达式 1/x 就肯定不会有这么高成本了。下面的例子是将 y 定义为一个严格的 val：

```
var x = 0
val y = try {
  1 / x
} catch {
  case _: Exception =>
    x = 1
    1 / x
}
println(y)
```

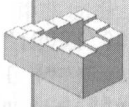
虽然这个代码也产生了预期的结果，但是这里你只有一次额外的机会可以正确地初始化 y（在 catch 块中）。而如果将 y 定义为 lazy val 则相反，每次访问未初始化的 lazy val 时都会检查初始化阶段直到初始化成功。

Lazy val 的这个特性在初始化所依赖的资源不能立即获得的情况会很有用。例如，假设在服务器启动期间，要生成某个文件，而生成文件所需要的数据得花些时间才能取到。

比如，假设你希望引进一个公共变量，这个变量依赖最终生成的文件内容，而你并不想一直阻塞一个线程直到文件内容写好。解决这个问题的一个简单的方法是将变量定义为 lazy val，简单地假设这个文件是可用的。

```
import io.Source
lazy val res = Source.fromFile("./processingresult.txt").
  getLines.filter(_.contains("quux"))
```

如果 res 在文件可用之前就被引用，它会在初始化期间抛出一个异常，并给调用代码发信号说数据集合还没有完成。一旦结果文件写完，res 就会正确地初始化为期望的值。



每次访问时 Scala 都会再次试图初始化 lazy val，直到初始化成功。这使它们对延迟的资源初始化非常有用。

第 12 章

集合的迭代顺序

在提供标准的函数式惯用语法（例如 for 表达式和 map，flatMap 的结合）的同时，Scala 也可以通过 for 循环和 foreach 方法支持集合上的命令式操作。

在这个谜题里，用了这两种操作（函数式和命令式）来按升序打印罗马数字符号。让我们看看在 REPL 命令行执行下面的代码会是什么结果吧。

```
case class RomanNumeral(symbol: String, value: Int)

implicit object RomanOrdering extends Ordering[RomanNumeral] {
  def compare(a: RomanNumeral, b: RomanNumeral) =
    a.value compare b.value

}

import collection.immutable.SortedSet

val numerals = SortedSet(
  RomanNumeral("M", 1000),
  RomanNumeral("C", 100),
  RomanNumeral("X", 10),
  RomanNumeral("I", 1),
  RomanNumeral("D", 500),
  RomanNumeral("L", 50),
  RomanNumeral("V", 5)
)

println("Roman numeral symbols for 1 5 10 50 100 500 1000:")
for (num <-numerals;sym = num.symbol) { print(s"${sym} ") }
numerals map { _.symbol } foreach { sym => print(s"${sym} ") }
```

可能的结果

1. 打印出：

```
Roman numeral symbols for 1 5 10 50 100 500 1000:
I V X L C D M
C D I L M V X
```

2. 打印出：

```
Roman numeral symbols for 1 5 10 50 100 500 1000:
M C X I D L V
M C X I D L V
```

3. 打印出：

```
Roman numeral symbols for 1 5 10 50 100 500 1000:
I V X L C D M
I V X L C D M
```

4. 打印出：

```
Roman numeral symbols for 1 5 10 50 100 500 1000:
C D I L M V X
I V X L C D M
```

解释

基于候选答案，你需要首先决定这个调用会按什么顺序迭代：用什么顺序将这些数字添加到集合里，是值的顺序还是符号的顺序呢？

因为你在处理一个排序的集合，所以可以假设是按排序的顺序进行迭代而不是声明的顺序。看上去结果正是按值排序实现了迭代。

那么，两个语句一定都会按照数字值升序排序迭代并打印出结果 `I V X L C D M` 吗？

并非如此。正确的答案是 1：

```
Roman numeral symbols for 1 5 10 50 100 500 1000:
scala> for (num <- numerals;sym = num.symbol) {
         print(s"${sym} ") }
I V X L C D M
```

```
scala> numerals map { _.symbol } foreach { sym =>
         print(s"${sym} ") }
C D I L M V X
```

为了开始理解这个结果，让我们先检查第一个语句，它的行为与预想一致。根据 Scala 语言规范，将简单的 `for (i <- expr){ fun(i) }`循环提糖成 `foreach` 方法调用是 `expr foreach { i => fun(i) }`[①]：

在这个例子中，由于附加的值定义[②]`sym = num.symbol`，这个 `for` 循环并非如此简单。为了能将 `num` 和 `sym` 传递进循环体，编译器在生成器中将 `numerals` 替换成了元组集合 `(num, sym)`。然后在这个元组集合上调用 `foreach` 并在将它们传递给实际循环体之前从元组中抽取这两个元素[③]。简而言之，`for` 循环提糖大致如下面所示：

```
numerals map { num =>
  val sym = num.symbol
  (num, sym)
} foreach { case (num, sym) =>
  println(sym)
}
```

输出结果太令人惊讶了，与第二个语句类似：

```
numerals map { num => num.symbol } foreach { ... }
```

因此，在这两个例子中，`foreach` 并非在原始的 `numerals` 集上调用，而是在 `map` 返回的结果上调用。反过来，这也解释了为什么产生了不同的结果。

那么 `map` 返回的结果是什么呢？Scala 集合的一个主要特征是转换会保持集合的类型不变，如 `map`。在这个例子里，结果就不会是与原始集合有着相同迭代顺序的任意集合，而是一个新的排序集 `SortedSet`。这个新的 `SortedSet` 的迭代顺序由它的元素来决定，而不是由原始集合中的元素顺序决定。

因为元组首先按它们的第一个元素来排序[④]，它们只是一系列 `numerals` 元素，`numerals` 的迭代顺序和元组的排序集是相同的。其结果就是：按期望的顺序打印出标识符，比如，按值的升序顺序。

然而，第二个语句中所创建的符号集合是一个字符串的排序集合，因此就是按字典顺序排序的。由于这个原因，第二个语句就是按照标识符的字母顺序打印，

---

① Odersky，《Scala 语言规范》，6.19 节。[Ode14]

② 同上。

③ 同上。

④ 参见 Scaladoc，scala.math.Ordering。[EPF]

而不是按照期望的值增加的顺序。

讨论

当然，在这个特定的例子中，你能简单地避免中间 map 和对 numerals 的直接迭代，避免非预期的重排序和对集合不必要的迭代：

```
scala> for (num <- numerals) { print(s"${num.symbol} ") }
I V X L C D M

scala> numerals foreach { num => print(s"${num.symbol} ") }
I V X L C D M
```

在那个通用例子中，可能的选择就是对原始集合的视图[①]应用转换：

```
scala> numerals.view map { _.symbol } foreach { sym =>
         print(s"${sym} ") }
I V X L C D M
```

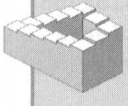
在视图上调用 map 不会创建一个中间的排序集，而是以 lazily 的方式应用 num => num.symbol，只有当要转换的 numerals 集合的下一个元素被检索时才打印在 foreach 循环中。

视图的迭代顺序与原始集合的顺序是一致的，所以就以我们期望的顺序打印出标识符来。

不过，Scala 2.11 对视图的使用仍然有极大的争议。一个简单的替代方法是从一个转换后不影响迭代顺序的集合类型开始，例如 Seq：

```
scala> numerals.toSeq map { _.symbol } foreach { sym =>
         print(s"${sym} ") }
I V X L C D M
```

通常附加的迭代成本会比原始集合高，但是使用起来却简单直观。



如果你正在一个集合上进行转换，尤其是当要串起多个操作时，注意，原始集合的迭代顺序可能会发生改变。如果你需要稳定不变的迭代顺序，就将原始集合转换成 sequence。

① 参见 Scala 集合库文档中的“视图”一节。[Odea]

第 13 章

自引用

在许多语言里，递归变量定义都是一个复杂的问题。最简单的递归变量定义就是引用它自己。下面的代码试图定义两个递归变量。让我们看看在 REPL 中执行下面的代码会是什么结果吧。

```
val s1: String = s1
val s2: String = s2 + s2

println(s1.length)
println(s2.length)
```

可能的结果

1. 打印出：

```
0
0
```

2. 两个 `println` 语句都抛出 `NullPointerException` 异常。

3. 第一个 `println` 语句抛出 `NullPointerException` 异常，第二个语句打印出：

```
8
```

4. 两个 `println` 语句都编译失败。

解释

你可能在想 Scala 中到底是否允许自引用呢。如果像第二个语句那样应用自引用，编译器也许只是抱怨一下但还是会执行。

或者，你也许认为自引用变量会用缺省值给变量赋值，这会导致 NullPointerException 异常。

正如所发生那样，两个语句的确都编译成功了。只不过其中一个抛出了预期的运行时异常：

```
scala> println(s1.length)
java.lang.NullPointerException
  ...
scala> println(s2.length)
8
```

出人意料的是：正确答案是 3。

在弄明白第二个例子中究竟发生了什么以前，让我们先看看 Scala 通常是如何对待这种递归定义的。根据 Scala 语言规范，递归值定义在 Scala 中的确是有效的[①]。唯一的条件是，必须要显式给出这个值的类型：

```
scala> val s = s
<console>:7: error: recursive value s needs type
       val s = s
               ^
```

当编译器评估表达式右边的赋值语句时，通常会给任何未初始化的变量赋予缺省值：

```
scala> val x = y; val y = 10
<console>:7: warning: Reference to uninitialized value y
       val x = y; val y = 10
                  ^


x: Int = 0
y: Int = 10
```

当出现任何明显未初始化的变量自身时，也会给它赋缺省值。因此，在本例中，会给 s1 赋值 null，这是 String 和 AnyRef 的缺省值：

```
scala> val s1: String = s1
<console>:7: warning: value s1 does nothing other than call
  itself recursively
       val s1: String = s1
                        ^


s1: String = null
```

① Odersky，《Scala 语言规范》，4.1 节。[Ode14]

那么 s2 呢？这里与 s1 相似，所有在声明中出现的 s2 都被替换成缺省值 null，所以它的初始化语句就是 null+null。

Scala 编译器将两个字符串常量的连接字符转换成字节码，等于：

```
String s2 = (new StringBuilder()).append(null)
            .append(null).toString();
```

这与 Java 的行为一致，正如 Java 虚拟机规范所描述的那样[①]。StringBuilder[②]则相反，将 null 的引用转换成字符串“null”。因而 s2 的值就是 8 个字符串“nullnull”：

```
scala> val s2: String = s2 + s2
s2: String = nullnull
```

讨论

本例中这些自引用定义都是不必要的，也是应该避免的。变量类型的缺省值可以用来代替任何自引用变量。这是在任何情况下编译器要做的事。在许多语言中，这种自引用甚至是不允许的。例如 Java 编译器就会报错：

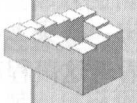
```
public class SelfRef {
  String s = s + s;

  public static void main(String[] args) {
    System.out.println(new SelfRef().s.length());
  }
}

$ javac SelfRef.java
SelfRef.java:2: error: selfreference in initializer
  String s = s + s;
             ^


SelfRef.java:2: error: selfreferencein initializer
  String s = s + s;
                 ^


2 errors
```

① Lindholm 等，《Java 虚拟机规范》，15.18.1.1 节。[Lin13]

② 参见 Javadoc，StringBuilder。[Ora]

不过，Java 中的自引用检查并不是非常成熟，很容易将编译器骗过去：

```
public class SelfRef2 {
  String s = this.s + this.s; // fool the compiler

  public static void main(String[] args) {
    System.out.println(new SelfRef2().s.length());
  }
}
$ javac SelfRef2.java

$ java SelfRef2
8
```



避免定义自引用变量：用变量类型代替程序中出现的定义了变量缺省值的变量。

第 14 章

Return 语句

与 Java 不同，Scala 并不要求必须有一个显式的 return 语句来返回值。如果漏掉 return，方法就会继承性地返回上一个表达式的结果。

尽管如此，显式的 return 语句偶而也会出现在 Scala 代码中。事实上，如下面这段程序所示，一个方法可以有多个 return 表达式。让我们看看这段程序会做什么吧！

```
def sumItUp: Int = {
  def one(x: Int): Int = { return x; 1 }
  val two = (x: Int) => { return x; 2 }
  1 + one(2) + two(3)
}

println(sumItUp)
```

可能的结果

1. 打印出：

```
3
```

2. 打印出：

```
4
```

3. 打印出：

```
6
```

4. 编译失败，并报出 unreachable code 错误。

解释

这段代码中的数字字面量立即引起了你的注意，就是出现在 return 语句之后的 1 和 2。看它们在代码中的位置，这行代码将永远无法访问到。编译器很可能会检测到这个问题并返回“unreachable”错误代码而失败，候选项 4 是正确答案，对吗？

不要如此匆忙做决定！Scala 语言规范规定：任何跟在 return 之后的表达式都不评估。而是简单地忽略[①]。因而无法访问到的字面量 1 和 2 变成了红鲱鱼。排除了这个选项之后，相信有一些常识就能帮助你找出正确答案了。one 方法和 two 函数值都返回传递给它们的参数，如分别是 2 和 3。因此，结果是 6，所以第 3 个候选答案一定是正确答案。

原因可以很容易在 REPL 中进行验证：

```
scala> println(sumItUp)
3
```

嗯，不是期望的结果 6。正确的答案是第 1 个选项！让我们回过头来再看看代码，看我们忽略了什么。由于 return 语句里 two 值的函数字面量定义看上去有些奇怪，return 关键字在函数体中和在方法中有相同的语义，这难道是错误的吗？REPL 再次成为你的朋友：

```
scala> val two = (x: Int) => { return x; 2 }
<console>:7: error: return outside method definition
       val two = (x: Int) => { return x; 2 }
```

的确，返回的结果不是我们所期望的！得再咨询咨询语言规范[②]了：

return 表达式 return e 一定要发生在某个封闭的命名方法或函数[f]程序体内，return 表达式评估表达式 e 并返回它的值作为函数 f 的结果。

第一个 return x 语句的封闭的命名方法是方法 one，不过，第二个 return x 语句的封闭的命名方法并不是函数值 two，而是方法 sumItUp。这就是这个谜题的本质。表面上看好像函数值 two 应该算得上是个封闭的命名方法，但是只有方法和本地函数（def）才能作为 return 语句的封闭范围。所以，当 3 被应用

① Odersky，《Scala 语言规范》，6.20 节。[Ode14]

② Odersky，《Scala 语言规范》，6.20 节。[Ode14]

到函数值 two 时，它立即被作为整个方法 sumItUp 的结果返回。方法 one 的调用结果则被忽略。

讨论

地道的 Scala 编程样式并不鼓励使用显式的 return 表达式，在特殊情况下，方法里也可以有多个 return 语句。这有助于编写更加简洁更少复杂性的方法。

由于不鼓励使用显式 return 语句，你可能会问，是否还有必要在 Scala 程序中使用 return 呢？毕竟，大多情况下显式的 return 语句都可以用相等表达式来代替。Scala 中几乎所有的控制结构都是表达式，所以这并不困难。例如，让我们看看下面这个方法：

```
def fibonacci(n: Int): Int = {
  if (n < 2) return n
  fibonacci(n -1)+ fibonacci(n -2)
}
```

这个方法可以写成没有 return 语句的形式，修改后的代码如下：

```
def fibonacci(n: Int): Int =
  if (n < 2)   n
  else fibonacci(n - 1)+ fibonacci(n - 2)
```

如果想从多个位置早点返回值，或者想检查检查函数的先决条件是否能满足，都可以用这种方式重写代码。这些情况可以实施成嵌套的 if-else 表达式。不过，这样做就会使正常的执行路径变得不那么明显了[①]。在这种情况下，可以用 return 语句跳出方法的执行从而使代码更具有可读性。此外，显式的 return 语句还能用来提高性能，比如，用 return 语句退出循环。

不过为了中断多级嵌套函数，return 表达式也是必需的。这种情况控制流必须从所有的嵌套函数跳出，进入到最内部的封闭方法（例如，return 作用的第一个方法，与最内部的封闭块相对）[②]。

假设你有一个这样的需求：对给定的一系列货币，通过查询 web services 来获得货币的汇率，返回自上次查询之后第一个汇率超过指定阈值的货币。下面是这个方法的实现：

---

① Fowler，《重构：改进现有代码的设计》。[Fow99]

② Griffith，《Scala 中 return 语句的目的》。[Gri10]

```
def findHotCurrency[A](currencies: Seq[A],
    threshold: (Double, Double) => Boolean): Option[A] = {
  for (currency <-currencies) {
    val oldRate = getCurrentRate(currency)
    val newRate = fetchRate(currency)
    if (threshold(oldRate, newRate)) return Some(currency)
  }
  None
}
```

注意，事实上这个例子中的 return 语句在一个嵌套函数的内部！也就是说，Scala 编译器转换了 for 表达式：

```
for (x <- expr) body
```

将上面的 for 表达式转换成：

```
expr foreach (x => body)
```

for 表达式的程序体变成了嵌套函数的程序体传递给 foreach。这也意味着如果你没有显式 return，如：

```
if (threshold(oldRate, newRate)) Some(currency)
```

尽管代码仍会编译，但是方法会返回不正确的结果 None，因为函数结果在传递给 foreach 时总是被丢弃了。

实施相同的控制流的另外一种方式是抛出并捕获异常。事实上，从嵌套的匿名函数中返回，是由抛出和捕获一个 scala.runtime.NonLocalReturnControl 异常来实现（由编译器）的。这就是为什么在 Scala 中捕获 java.lang.Throwable。Throwable 不是个好主意：这样做可能会干预用在函数字面量上的 return 语句的控制流。一般而言，异常处理是一个高代价的操作。基于这个原因，在性能关键的代码中要尽可能避免将 return 语句用在转换函数中。

最后，不可访问的代码 unreachable code（第 4 个候选项）实际上只是一个与模式匹配上下文相关的警告，模式能匹配任何东西，从而阻止其后的语句被访问[①]。

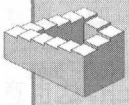
缺省情况下，Scala 编译器不会抱怨其他 unreachable code。如果需要，-Ywarn-dead-code 编译器选项能给你发出警告信息：

```
scala> def sumItUp: Int = {
         def one(x: Int): Int = { return x; 1 }
```

① 有关模式匹配更详细的讨论，请参见第 2 章。

```
        val two = (x: Int) => { return x; 2 }
        1 + one(2) + two(3)
      }
<console>:8: warning: dead code following this construct
        def one(x: Int): Int = { return x; 1 }
                                         ^
<console>:9: warning: dead code following this construct
        val two = (x: Int) => { return x; 2 }
                                        ^
sumItUp: Int
```

你可以进一步研究并使用-Xfatal-warnings 选项，使用这个选项后如有任何警告都会引起编译失败。



你应该尽可能避免使用显式的 return 语句。如果确实需要使用，也要知道上下文并确保可以在你需要的地方继续执行。

请记住，return 语句只会从方法和本地函数返回，而不会从定义在里面的函数值返回。

第 15 章

偏函数中的_

Scala 相当广泛地使用下划线（_）作为通配符标识。以下程序重点研究这个标识符的两种用法。让我们看看下面的程序会打印出什么结果吧！

```
var x = 0
def counter() = { x += 1; x }
def add(a: Int)(b: Int) = a + b
val adder1 = add(counter)(_)
val adder2 = add(counter) _

println("x = " + x)
println(adder1(10))
println("x = " + x)
println(adder2(10))
println("x = " + x)
```

可能的结果

1. 打印出：

```
x = 1
12
x = 2
11
x = 2
```

2. 打印出：

```
x = 1
11
x = 1
12
x = 2
```

3. 打印出：

```
x = 0
11
x = 1
12
x = 2
```

4. 打印出：

```
x = 2
11
x = 2
12
x = 2
```

解释

在这段代码中，定义 counter，adder1 和 adder2 的这几行代码是关键，所以让我们重点看看如何完全理解这几行代码。Counter 方法的定义含有一个副作用的例子。当评估 x 时，x 的当前值加 1 返回，因为在 Scala 中，方法的最后表达式就是它的返回值。

定义 adder1 和 adder2 的表达式似乎更加有趣。表面上看它们很相似，而且生成的函数值也有相同的类型：

```
scala> val adder1 = add(counter)(_)
adder1: Int => Int = <function1>
scala> val adder2 = add(counter) _
adder2: Int => Int = <function1>
```

然而，它们的语义却完全不同。在 adder1 中，表面上看好像其函数值是由偏函数 add 创建的。实际上这里的下划线_是个匿名函数占位符[①]。例如，请看下面这个表达式，这是一个占位符语法的例子：

```
_ + 1
```

它与下面的这个匿名函数是等价的：

```
x => x + 1
```

定义 adder1 的表达式也是相似的：

```
add(counter)(_)
```

---

① Odersky，《Scala 语言规范》，6.23 节。[Ode14]

可以将它扩展成：

```
a => add(counter)(a)
```

这是一个函数值，所以对参数 counter 的评估被延迟到评估 adder1 时。

定义 adder2 的表达式中的_则不同：

```
val adder2 = add(counter)_
```

这个表达式是个偏函数，因为没有给 add 方法提供参数 b，从而引发 eta expansion[①]将方法转换成其余参数的函数。这就很有必要为所提供的每个方法参数创建刷新的值，结果就是会立刻评估参数。在 adder2 的情况，编译器在场景后台对这些行生成以下内容：

```
val adder2 = {
  val fresh = counter()
  a => add(fresh)(a)
}
```

因此，adder1 和 adder2 之间的主要差别是如何评估参数 counter。在 adder1 中，每次使用 adder1 的时候都要对 counter 评估，而 adder2 中，counter 只评估一次，函数被调用的时候不用再次评估。

Eta expansion

Eta expansion[①]是将一个方法强制转换成一个对等的函数[②]的自动操作：

```
def foo[A, B](a: A): B
```

eta expansion 在以下条件满足时会执行[③]：

- 用下划线代替参数列表：

```
val f = foo _ // f is inferred as A => B
```

- 没有提供参数列表，但有期望的函数类型：

```
val f: A => B = foo
```

① Odersky，《Scala 语言规范》，6.26.5 节。[Ode14]

② Gleichmann，《函数式 Scala：将方法转变成函数》。[Gle11]

③ Zaugg，《在 Scala 中，为什么没有显式地指定其参数类型就不能部分应用一个函数？》。[Zau10a]

现在让我们再浏览一下程序。在前面三行，x 的值是 0：

```
var x = 0
def counter() = { x += 1; x }
def add(a: Int)(b: Int) = a + b
```

接着的下一行，定义了 adder1 函数值，counter 还没有评估，所以 x 仍然是 0：

```
val adder1 = add(counter)(_)
```

再接下来的一行当 adder2 初始化时 counter 方法首次被执行：

```
val adder2 = add(counter) _
```

因此，x 增加到 1。此时，为 adder1 提供的 add 方法的 a 参数被绑定到值 1。接着，x 的当前值被打印出来，此时仍然是 1：

```
println("x = " + x)
```

随后的行应用 adder1(10)，执行 counter 并将 x 增加到 2：

```
println(adder1(10))
```

结果是打印出 12，接下来的语句表明 x 等于 2。再接着的行应用 adder2（10）：

```
println(adder2(10))
```

函数 adder2 创建的时候，counter 评估为 1。它会调用 adder2 执行 10+1 产生结果 11。最后一行将打印 2，因为 x 值自从上次打印之后还没有变。

因此，正确的答案是 1：

```
scala> println("x = " + x)
x = 1
scala> println(adder1(10))
12
scala> println("x = " + x)
x = 2
scala> println(adder2(10))
11
scala> println("x = " + x)
x = 2
```

随后对 adder2 的每次调用都会返回相同的结果，而调用 adder1 则不同，每次调用都持续增加 x：

```
scala> println(adder1(10))
13
```

```
scala> println(adder2(10))
11
```

讨论

这个程序并不值得说太多。首先，使用 var 没有反映 Scala 语言规范。即使 Scala 支持可变声明，但它还是鼓励函数式编程，如，首选 val 和不变性。

其次，这个令人惊讶的行为只在有副作用的代码中才会显现出来。counter，x+=1 的副作用使这个程序很难解释其原因。这是命令式编程的一个缺点：当传递和操纵函数的时候，很难知道它们什么时候被调用、调用了多少次。当调用一个对外部世界没有副作用的函数时，其结果会更加可预测。

最后，一个普遍的错误是：原本打算用偏应用方法 add 构建 adder，但却没有显式写出下划线标识符，如下所示：

```
val adder3 = add(counter)
```

这样写会导致编译错误：

```
scala> val adder3 = add(counter)
<console>:10: error: missing arguments for method add;
  follow this method with '_' if you want to treat it
  as a partially applied function
      val adder3 = add(counter)
                       ^
```

这个错误的原因是编译器不会显式地期望 adder3 是个函数值（例如，Function1[Int, Int]类型）。如果指定了 adder3 的类型，编译器就会执行 eta expansion 而不会报错了。

```
scala> val adder3: Int => Int = add(counter)
adder3:Int => Int => <function1>
```

当一个方法期望用函数作为参数的时候，这种方法就派上用场了。

例如，考虑在 List[A]上使用 folder 方法：

```
def fold[A1 >: A](z: A1)(op: (A1, A1) => A1): A1
```

假设要用 sum 操作来执行 fold，sum 定义如下：

```
def sum(a: Int, b: Int) = a + b
```

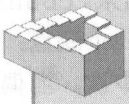
你可以显式地给 sum 传递一个下划线：

```
List(1, 2, 3).fold(0)(sum _)
```

也可以省略下划线：

```
List(1, 2, 3).fold(0)(sum)
```

简而言之，当 Scala 知道它期望的类型是一个函数且函数的类型与方法的类型一致时，就可以在方法名后省去下划线。



请熟练掌握匿名函数和偏函数以及下划线标识符的其他用法。

总的来说，就是要努力避免写那些容易让读者曲解的代码。

第 16 章

多参数列表

通过克里化过程，Scala 方法可以定义多个参数列表。让我们看看下面两个方法的定义：

```
def regular(x: Int, y: Int, z: Int) = x + y + z
def curried(x: Int)(y: Int)(z: Int) = x + y + z
```

如果给它们相应的参数赋一样的值，这两个方法得出的结果完全相同：

```
scala> regular(1, 2, 3)
res0: Int = 6

scala> curried(1)(2)(3)
res1: Int = 6
```

尽管有这种相似性，但克里化方法与常规方法还是有根本的不同：调用克里化方法代表了三次连续的函数调用[①]。

Scala 也支持缺省参数，这极大地减少了方法重载的需求。此外，通过将缺省参数与命名参数绑定，当调用一个方法时，可以按任何顺序提供参数子集。这就可以更加灵活地调用方法，也能使调用程序的站点对重构错误更加鲁棒。

下面的程序定义了一个有两个参数列表的克里化方法。第二个列表中的参数规定了缺省参数值。

让我们看看在 REPL 命令行执行下面的代码会是什么结果呢？

```
def invert(v3: Int)(v2: Int = 2, v1: Int = 1) {
  println(v1 + ", " + v2 + ", " + v3)
}
val invert3 = invert(3) _
```

① 如果提供了所有的参数，编译器能将这些连续的调用转成单个方法调用。

```
invert3(v1 = 2)
invert3(v1 = 2, v2 = 1)
```

可能的结果

1. 打印出：

```
2, 2, 3
2, 1, 3
```

2. 第一次调用 invert3 编译失败，第二次调用打印出：

```
1, 2, 3
```

3. 第一次调用 invert3 编译失败，第二次调用打印出：

```
2, 1, 3
```

4. 第一次调用 invert3 编译失败，第二次调用打印出：

```
2, 2, 3
```

解释

invert3 是个函数类型，因为 invert 调用只提供了第一个参数：

```
scala> val invert3 = invert(3) _
invert3: (Int, Int) => Unit = <function2>
```

函数 invert3 有两个 Int 类型的参数，但第一次调用只提供了一个：v1 = 2。参数 v2 有个缺省值 2，所以你可能认为第一次调用 invert3 会打印出：

```
2, 2, 3
```

第二次调用打印出：

```
2, 1, 3
```

让我们验证一下：

```
scala> invert3(v1 = 2)
<console>:10: error: not enough arguments for method apply:
  (v1: Int, v2: Int)Unit in trait Function2.
```

```
Unspecified value parameter v2.
              invert3(v1 = 2)
                      ^


scala> invert3(v1 = 2, v2 = 1)
1, 2, 3
```

所以，不是第 1 个候选答案。这表明正确的答案是 2！到底发生了什么呢？

编译器错误给了我们一个提示：答案在于理解函数类型实际上是如何实现的。如 Scala 语言规范所述[①]，函数类型都有一个 apply 方法，定义如下[②]：

```
// abstract apply method of Function2[T1, T2, R]
def apply(v1: T1, v2: T2): R
```

因此，在对这些行进行编译时，原始的程序被扩展成如下行：

```
def invert(v3: Int)(v2: Int = 2, v1: Int = 1) {
  println(v1 + ", " + v2 + ", " + v3)
}

def invert3 = new Function2[Int, Int, Unit] {
  def apply(v1: Int, v2: Int): Unit = invert(3)(v1, v2)
}

invert3.apply(v1 = 2)
invert3.apply(v1 = 2, v2 = 1)
```

关于 apply 方法的明显特征有如下两点比较突出。第一，方法参数没有缺省值。因此，第一次调用 invert3 就引起了编译器错误，因为只传递了一个参数 v1=2。

第二，参数名字实际上是 v1 和 v2。如果 vert 方法使用了不同的参数名，那么第二次调用 invert3 也会编译失败。Function2 的所有实例的 apply 方法都用这个参数名字，并没有关联到初始的 invert 方法的参数名。

第三，因为 invert3 恰好有相同的参数名，但顺序相反，所以第二次调用打印出：

```
scala> invert3(v1 = 2, v2 = 1)
1, 2, 3
```

---

① Odersky，《Scala 语言规范》，3.2.9 节。[Ode14]

② 参见 Scaladoc Function2。[EPF]

即便你可能认为命名参数附属于 invert，但实际上它们指向 apply 方法的参数。准确地说，apply 的 v1 参数与 invert 的 v2 一致被设为 2。同样地，apply 方法的 v2 参数与 invert 的 v1 一致被设为 1。

讨论

你可能会问在单独的参数列表中定义 invert 方法的参数其基本原理是什么呢。之所以用这种方式而不是用单一参数列表的方式来定义方法参数，是因为这种方式有多种优点。首先，定义了缺省参数时你能引用以前的参数列表的参数：

```
def area(x: Int)(y: Int = x) = x * y
```

其次，多个参数列表也能有助于类型推断，因为早参数列表中推断出的类型并不需要指定。以 Scala 集合库中的 foldLeft 方法[①]举例：

```
def foldLeft[B](z: B)(op: (B, A) => B): B
// no need to specify Int for B
Seq("I", "II", "III").foldLeft(0)(_ + _.length)
```

此外，多个参数列表允许你可以有隐式和非隐式参数：

```
def maxBy[B](f: A => B)(implicit cmp: Ordering[B]): A
```

多参数列表还可以用来定义将参数列表放在大括号内而不是圆括号内的流畅的 API[②]：

```
def benchmark(times: Int)(block: => Unit): Unit
benchmark(10000) {
  ...
}
```

由这道谜题，我想说的克里化的最后一个优点是：多个参数列表能使偏函数表达得更加简洁。例如，如果将 invert 方法声明为使用单一参数列表的形式：

```
def invert(v3: Int, v2: Int = 2, v1: Int = 1) {
  println(v1 + ", " + v2 + ", " + v3)
}
```

那就不能再用下面这种语法只用一个参数来调用这 invert 方法了。

---

① 关于多参数列表中发生的类型参数的进一步讨论，请参见第 6 章。

② 关于大括号与圆括号比较的更多详细讨论，请参见第 35 章。

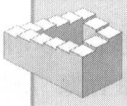
```
scala> def invert3 = invert(3) _
<console>:8: error: _ must follow method; cannot follow Unit
       def invert3 = invert(3)
                             ^
```

你必须要为每个省略的参数提供下划线标识符和类型归属：

```
scala> def invert3 = invert(3, _: Int, _: Int)
invert3: (Int, Int) => Unit
```

说到使用单一参数列表的方法，我们唯一能不提供参数类型还能成功的情况是：当我们为所有的参数传递一个下划线标识符的时候。

```
scala> def invert3 = invert(_, _, _)
invert3: (Int, Int, Int) => Unit
```



记住，当调用偏函数时，命名参数并不能解析原始的方法，但能解析生成的函数对象。你能通过不让 Scala 函数特质用参数名来避免这个问题。

第 17 章

隐式参数

能够创建和分发偏函数，有助于你从特定的功能转向更加通用和可重用的功能。比如，我们可以写一个通用的 add 方法，再用这个通用方法为用户用例客制化，而不是用写死的代码固定用 2 增加值的方法。

```
def add(x: Int)(y: Int) = x + y
def addTo2 = add(2) _

scala> addTo2(3)
res2: Int = 5
```

Scala 也支持 implicit，这就可以从上下文中挑选出参数，而不是显式地传递这些参数给方法和函数。

这两个是如何一起做到的呢？让我们看看在 REPL 中执行下面的代码会是什么结果吧！

```
implicit val z1 = 2
def add(x: Int)(y: Int)(implicit z: Int) = x + y + z
def addTo(n: Int) = add(n) _

implicit val z2 = 3
val addTo1 = addTo(1)
addTo1(2)
addTo1(2)(3)
```

可能的结果

1. 打印出：

```
5
6
```

2．打印出：

```
6
6
```

3．第一次调用 addTo1 打印出：

```
5
```

第二次调用编译失败。

4．第一次调用 addTo1 编译失败，第二次调用打印出：

```
6
```

解释

你可能在想是否 add 的隐式参数会变成由 addTo 返回的偏函数的一个隐式参数呢，或者是否会以某种方式将它转换成常规的参数列表呢。是否 addTo1 有一个隐式参数，或者编译器是否会试图解析 addTo 调用内的隐式参数，所以对 z 用模糊的隐式值会有问题呢？

都不对。事实上，正确的答案是 3：

```
scala> addTo1(2)
res0: Int = 5

scala> addTo1(2)(3)
<console>:12: error: Int does not take parameters
              addTo1(2)(3)
                     ^
```

add 方法的隐式参数发生了什么呢？如何避免 z1 和 z2 的两个模糊的隐式值之间发生冲突呢？

为了解决第一个问题，首先观察 addTo1 的类型特征，可以看出这是只有一个参数的函数，而非两个参数：

```
scala> val addTo1 = addTo(1)
addTo1: Int => Int = <function1>
```

换句话说，add 的常规参数 y 也是 eta-expanded 函数 addTo1 的一个参数，而隐式参数 z 不是。隐式参数是在方法体 addTo 内应用 eta expansion[①]

① Eta expansion 在第 15 章中会更详细的讨论。

时解析的。

然而，编译器是如何选择隐含值 z1 的呢？毕竟，addTo 在表达式 val addTo1 =addTo(1)中被调用时，z1 和 z2 都在范围内。

解释是直观的：隐式参数是由编译器在方法 addTo 编译时解析的，而不是在稍后调用时解析的[①]，此时，只有隐式参数 z1 在范围内。addTo 方法实际上就变成：

```
def add(x: Int)(y: Int) = x + y + 2
def addToWithResolvedImplicit(n: Int) = add(n) _
```

其表现出的行为与原来的 addTo 方法相同：

```
scala> val addTo1WithResolvedImplicit =
         addToWithResolvedImplicit(1)
addTo1WithResolvedImplicit: Int => Int = <function1>

scala> addTo1WithResolvedImplicit(2)
res2: Int = 5

scala> addTo1WithResolvedImplicit(2)(3)
<console>:12: error: Int does not take parameters
              addTo1WithResolvedImplicit(2)(3)
                                            ^
```

讨论

如果声明 addTo 时，z1 和 z2 的隐式值都在范围内，代码确实会编译失败。

```
implicit val z1 = 2
implicit val z2 = 3

def add(x: Int)(y: Int)(implicit z: Int) = x + y + z

scala> def addTo(n: Int) = add(n) _
<console>:10: error: ambiguous implicit values:
 both value z1 of type => Int
 and value z2 of type => Int
 match expected type Int
       def addTo(n: Int) = add(n) _
                           ^
```

① Odersky，《Scala 语言规范》，7.2 节。[Ode14]

用 eta expansion 并不能阻止隐式参数的解析，但可以完全避免 eta expansion，从而保留 z 这一个参数（只是显式地构造一个匿名函数返回）：

```
def addToReturnAnonFun(n: Int) =
  (y: Int) => (z: Int) => add(n)(y)(z)

scala> val addTo1ReturnAnonFun = addToReturnAnonFun(1)
addTo1ReturnAnonFun: Int => (Int => Int) = <function1>

scala> addTo1ReturnAnonFun(2)
res0: Int => Int = <function1>

scala> addTo1ReturnAnonFun(2)(3)
res1: Int = 6
```

addToReturnAnonFun 方法创建并返回有两个参数列表的克里化函数，与 add 方法的 lists(y) 和 lists(z) 这两个参数列表一致。如果用占位符语法使匿名函数的定义更加简洁，你就能得到一个有两个参数的函数，而不是一个克里化函数。

匿名函数和隐式参数

你可能理所当然地认为隐式参数是在 eta expansion 期间解析的，以为匿名函数不支持隐式参数。事实上它们支持[①]。

不过匿名函数里隐式参数的语义与方法里的隐式参数略有不同：编译器不会自动解析匿名函数里的隐式参数（调用匿名函数时需要显式指定）。

那么匿名函数里的隐式参数与常规参数的操作有什么不同呢？隐式关键字 implicit 的作用就是使匿名函数体内的隐式解析更加合适。

一个简短的例子可以证明这个：

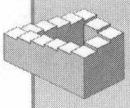
```
def iNeedAnImplicit(implicit n: Int) = n + 1

scala> val anonFun = { x: Int => y: Int =>
         x + y + iNeedAnImplicit }
<console>:8: error: could not find implicit value for
  parameter n: Int
       val anonFun = { x: Int => y: Int =>
         x + y + iNeedAnImplicit }
```

① Ddersky，《Scala 语言规范》，6.2.3 节。[Odel 4]

```
                    ^

scala> val anonFunWithImplicitParam = { x: Int =>
         implicit y: Int => x + y + iNeedAnImplicit }
anonFunWithImplicitParam:
  Int => (Int => Int) = <function1>

implicit val z = 2

scala> anonFunWithImplicitParam(1)(2)
res4: Int = 6
// the compiler will not supply the implicit argument
scala> anonFunWithImplicitParam(1)
res5: Int => Int = <function1>
```

```
def addToReturnPlaceholderAnonFun(n: Int) =
  add(n)(_: Int)(_: Int)

scala> val addTo1ReturnPlaceholderAnonFun =
         addToReturnPlaceholderAnonFun(1)
addTo1ReturnPlaceholderAnonFun: (Int, Int) => Int = <function2>
```

在这两个例子中，偏函数的第二个参数不是隐式的，调用该函数时必须显式提供这个参数。



记住，隐式参数是在 eta expansion 期间解析的，它们并不是能使函数得到结果值的参数。

第 18 章

重载

重载方法可以用来提供可以使用不同参数组合的函数，是一种常用的方式。这里必须得小心点儿。因为一个重载方法的新版本意味着编译器调用方法时需要区别多个可应用的版本。如果编译器不能识别唯一的最可应用的方法就会编译失败：

```
def foo(n: Int, a: Any) {
  println(s"n: ${n}, a: ${a}") }

scala> foo(1, 2)
n: 1, a: 2

object A {
  def foo(n: Int, a: Any) {
    println(s"n: ${n}, a: ${a}") }
  def foo(a: Any, n: Int) {
    println(s"a: ${a}, n: ${n}") }
}

scala> A.foo(1, 2)
<console>:10: error: ambiguous reference to
  overloaded definition,
both method foo in object A of type (a: Any, n: Int)Unit
and method foo in object A of type (n: Int, a: Any)Unit
match argument types (Int,Int)
              A.foo(1, 2)
                    ^
```

在下面的例子中，方法显然是为了阻止这种模糊的方法调用而设计的。让我们看看执行下面的代码结果是什么。

```
object Oh {
  def overloadA(u: Unit) = "I accept a Unit"
  def overloadA(u: Unit, n: Nothing) =
      "I accept a Unit and Nothing"
```

```
  def overloadB(n: Unit) = "I accept a Unit"
  def overloadB(n: Nothing) = "I accept Nothing"
}

println(Oh overloadA 99)
println(Oh overloadB 99)
```

可能的结果

1. 第一个语句编译失败，第二个语句打印出：

```
I accept Nothing
```

2. 第一个语句打印出：

```
I accept a Unit
```

第二个语句编译失败。

3. 两个语句都编译失败。

4. 打印出：

```
I accept a Unit
I accept a Unit
```

解释

因为 Int 并不是从 Unit 继承来的，你可能会怀疑是第一个还是两个都会编译失败。或者你可能认为编译器能将参数 99 当作 Unit 处理，并推导出两种情况都打印出“I accept a Unit”。但是你确定第一种情况编译器不能接受 Int 为 Unit，第二种情况编译器也会拒绝吗？

哦，不，第一个语句能接受 Int 为 Unit。正确的答案的确是 2：

```
scala> println(Oh overloadA 99)
<console>:9: warning: a pure expression does nothing
  in statement position; you may be omitting necessary
  parentheses
              println(Oh overloadA 99)
                                   ^
```

```
I accept a Unit

scala> println(Oh overloadB 99)
<console>:9: error: overloaded method value overloadB
    with alternatives:
  (n: Nothing)String <and>
  (n: Unit)String
cannot be applied to (Int)
             println(Oh overloadB 99)
                             ^
```

对这个结果的解释是：第二个语句没有失败是因为有一个可应用的重载方法加进来，产生了一个模糊的引用。而第二个语句失败了，是因为 `overloadB(n: Nothing)` 方法不接受 `Int` 参数。与第一种情况不同的是，因为在方法调用的地方没有找到可应用的重载方法，所以原本可应用的方法此时也不再能应用了。

为了解释这个似乎很不寻常的行为，你需要深入了解 Scala 的重载解决方案[①]。如果有一个像 `overloadA` 的标识符引用一个类的多个成员，Scala 会应用一个两阶段算法来决定，如果可能就调用合适的成员。在第一阶段，编译器把成员声明的“形状”与调用比较，看看哪种“形状”可能看上去是正确。大致说来，编译器审视每个重载选项的参数数目，并与调用进行比较[②]。如果刚好有一个选项是可行的，就选择它。

在第一种情况中，这个可以让编译器避免潜在昂贵的基于类型解析的快捷方式是足够的：只有 `overloadA` 这一个可选的重载有“正确的形状”（例如，只有一个参数）。所以编译器就选择这个选项。但是仍然有一个障碍：如何让 Int 参数 99 能与期望的 `Unit` 类型匹配呢？

这里，Scala 的值转换[③]起了作用，编译器可以应用这些隐式转换在需要时将一个值类型转换成期望的不同类型。`overloadA` 的例子应用了值抛弃：编译器通过将它嵌入到 `{ 99; () }` 中将 `Int` 转换成期望的 `Unit` 类型。这意外地导致观察到的编译器警告。

然而 `overloadB` 的情况，基于形状的解析还不能充分决定唯一的选择，因为两个选项都只有一个参数。编译器因此继续进入到重载解析的第二个阶段，试

① Odersky，《Scala 语言规范》，6.26.3 节。[Ode14]
② Odersky，《Scala 语言规范》，6.26.3 节。[Ode14]
③ Odersky，《Scala 语言规范》，6.26.1 节。[Ode14]

图用最特定的参数类型发现一个精确的选择[①]。

这就是对观察到的行为的解释。因为基于类型的解析阶段的第一步是先判断所提供参数的类型，而不考虑可能要调用的方法期望什么类型。在这个例子中，这个过程等同于评估 `val arg = 99`（与 `val arg: Unit = 99` 相对）并查看结果类型。这似乎是足够合理的了，因为此时编译器还不知道会调用哪个方法，所以也不知道期望什么类型。

因为判定参数 99 是一个 `Int` 类型，现在编译器继续试图识别一个最特定的可以应用这个参数的重载方法。但是 `Int` 既不是从 `Unit` 也不是从 `Nothing` 继承的，所以没有可以应用的选择！

此外，`overloadA` 将 99 转成 `Unit` 的值转换也不可用，因为它只有在期望的类型值是已知时才可应用。但是因为编译器还不能识别一个要调用的最特定的方法，也就不知道必须的期望类型。因此两个选择都不可应用，所以代码就编译失败了。

Magnet 模式

如这里所证实的那样，Scala 的方法重载有某种缺点[②]，因而不能用于这个谜题的代码中，Magnet 模式[③]提供了一个有趣的选择。在 Magnet 模式中，多个重载方法定义被替换为一个单个方法和一套对每个支持的参数类型组合的隐式转换。

讨论

这个问题的一个变种很可能发生在生产代码中，它涉及方法的参数列表为空的情况。在期望这种类型的地方可以通过 `eta expansion` 转换到 `() => T` 函数类型，但是在基于类型的重载解析期间，编译器不考虑这种转换。

```
object Oh2 {
  def nonOverloadedA(f: () => Any) =
    "I accept a noarg function"
```

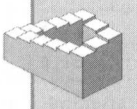
① Odersky，《Scala 语言规范》，6.26.3 节。[Ode14]

② Zaugg，《为什么要避免方法重载？》。[Zau10b]

③ Mathias，《Magnet 模式》。[Mat12]

```
  def overloadedA(f: () => Any) =
    "I accept a noarg function"
  def overloadedA(n: Nothing) = "I accept Nothing"
}

def emptyParamList() = 99

scala> Oh2 nonOverloadedA emptyParamList
res0: String = I accept a noarg function

scala> Oh2 overloadedA emptyParamList
<console>:10: error: overloaded method value overloadedA
    with alternatives:
    (n: Nothing)String <and>
    (f: () => Any)String
  cannot be applied to (Int)
              Oh2 overloadedA emptyParamList
                  ^
```

对于这个潜在的问题该怎么处理呢？因为错误仅发生在方法调用的地方，作为代码的作者，你除了避免重载完全相同形状的方法以外也没有别的选择。作为方法的调用者，你也没有办法，只有保持对隐式类型转换敏锐的关注。可惜现在还没有编译器选项能让应用程序自动标记出这个问题。



1. 要提防由编译器插入的依赖期望类型的类型转换，例如丢弃值、视图应用（如应用隐式转换）和 eta expansion。如果加入了被调用方法的重载版本，这些转换可能就不再可用了。

2. 注意：定义与已经存在的重载方法形状相同的重载方法，无论这个新方法的参数类型是什么，都会导致调用代码编译失败。

第 19 章

命名参数和缺省参数

依赖参数列表中数据项的准确顺序的代码是脆弱的，尤其是如果参数类型都相同的时候：

```
def inEcosystem(predator: String, prey: String) {
  println(s"${predator} eat ${prey}")
}

scala> inEcosystem("cats", "mice")
cats eat mice

def inEcosystem(prey: String, predator: String) {
  println(s"${prey} are eaten by ${predator}")
}

// no idea that the definition has changed...
scala> inEcosystem("cats", "mice")
cats are eaten by mice

scala> inEcosystem(predator = "cats", prey = "mice")
mice are eaten by cats
```

Scala 也支持缺省参数[①]，这使它不必用爆炸式组合的重载方法就很容易写出通用的函数并提供通用的用户用例：

```
object MakeSequences {
  def mkSeq(end: Int, start: Int = 1, step: Int = 1) = ...
  // rather than...
  def mkSeq2(end: Int, start: Int, step: Int) = ...
  def mkSeq2(end: Int, start: Int) = mkSeq2(end, start, 1)
  def mkSeq2(end: Int) = mkSeq2(end, 1, 1)
}
```

① 有关更详细的缺省参数的讨论，请参见第 16 章。

这个谜题使用命名参数和缺省参数的组合来调用一个重载方法。让我们看看在 REPL 中执行下面的代码结果会是什么呢？

```
class SimpleAdder {
  def add(x: Int = 1, y: Int = 2): Int = x + y
}
class AdderWithBonus extends SimpleAdder {
  override def add(y: Int = 3, x: Int = 4): Int =
    super.add(x, y) + 10
}

val adder: SimpleAdder = new AdderWithBonus
adder add (y = 0)
adder add 0
```

可能的结果

1. 打印出：

```
14
12
```

2. 打印出：

```
14
14
```

3. 打印出：

```
13
14
```

4. 打印出：

```
11
12
```

解释

你可能想知道将 adder 声明为 SimpleAdder 在某种程度上是否会导致应用 add 方法的缺省值从而打印出 11 和 12 呢。另外一种可能是，如果选择 AdderWithBonus 的重载版本是否一定会打印出 14 和 14 呢？

并非如此。正确的答案是 3：

```
scala> adder add (y = 0)
res10: Int = 13

scala> adder add 0
res11: Int = 14
```

给出的这个可能的值，只有当两种情况都使用了 AdderWithBonus 类中 x 的缺省值（例如，即使指定 y=0 时）时才能得到这个结果，这是得到这个结果的唯一方式。

怎么会是这个结果呢？为了理解这个结果，你需要检查 Scala 是如何处理命名参数和缺省参数的[①]。因为 JVM 本身并不支持这种调用，编译器需要把用这种参数的调用转换成“常规”调用，即所有的参数都按函数定义所要求的顺序传递。

编译器通过为每个函数所要求的参数定义一个变量并将所有提供的参数（位置的和命名的）的值赋给合适的变量来完成。对于所有其他参数，编译器调用特定的自动增加到类定义缺省值的[②]“缺省方法”。然后用所有的参数调用目标方法。第二个语句中的 adder add 0 就转换成下面的代码：

```
{
  val x$1 = 0 // arg at position 0
  val x$2 = adder.add$default$2
  adder.add(x$1, x$2)
}
res12: Int = 14
```

在 adder add (y = 0)的情况，还需要另外一个步骤。编译器需要重新安排变量以便正确的值出现在期望的位置。当编译这个调用时用唯一可用的信息来实现：被调用的值的类型的方法定义。

因为 adder 的类型是 SimpleAdder（不是 AdderWithBonus，这是它的运行时类型），这意味着编译器将 0 值移动到 SimpleAdder 的 add 定义上参数 y 的位置。

```
{
  val x$1 = 0 // named arg for 'y'
  val x$2 = adder.add$default$1
  adder.add(x$2, x$1) // matches order in SimpleAdder.add
```

① Odersky，《Scala 语言规范》，6.6.1 节。[Ode14]

② Odersky，《Scala 语言规范》，4.6 节。[Ode14]

```
}
res13: Int = 13
```

缺省参数的值是通过调用缺省方法在运行时决定的（这里是 add$default$1）。AdderWithBonus 中指定的新的缺省值引起编译器为 SimpleAdder 中的方法生成重载。结果，就使用了为 AdderWithBonus 定义的缺省参数值，这是 adder 的运行时类型。

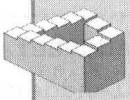
讨论

如果 adder 的类型不是显式指定的，结果就会少些意外：

```
val adder2 = new AdderWithBonus

scala> adder2 add (y = 0)
res14: Int = 14

scala> adder2 add 0
res15: Int = 14
```

很显然，避免主代码例子中的场景的最简单方式是确保 AdderWithBonus.add 中的参数顺序与重载版本的参数顺序一致。

```
class SimpleAdder {
  def add(x: Int = 1, y: Int = 2): Int = x + y
}
class AdderWithBonus2 extends SimpleAdder {
  override def add(x: Int = 4, y: Int = 3): Int =
    super.add(x, y) + 10
}

val adder3: SimpleAdder = new AdderWithBonus2
scala> adder3 add (y = 0)
res16: Int = 14
scala> adder3 add 0
res17: Int = 13

val adder4 = new AdderWithBonus2
scala> adder4 add (y = 0)
res18: Int = 14

scala> adder4 add 0
res19: Int = 13
```



1. 重载方法时要尽可能维持参数的顺序不变。

2. 当使用命名参数和缺省参数时，记住（对 Josh Suereth 原话的解释）：“名字是在编译时，值是在运行时。”

第 20 章

正则表达式

你可能听过一个很老的笑话："一个开发人员有个问题，决定用正则表达式来解决。这下好了，现在他们就有两个问题了。"正则表达式确实能快速把情况搞得更复杂。但是它们也的确非常强大，如果用得明智的话可以非常有用。

Scala的`scala.util.matching.Regex`类为正则表达式提供了实用函数。它的`findAllIn`方法返回`MatchIterator`，它能对一个字符串上出现的所有的正则表达式进行迭代：

```
scala> for (reMatch <-"l".r.findAllIn("I love Scala"))
         println(reMatch)
l
l
```

在下面的例子中，两次用`start`方法对`MatchIterator`查询字符串中首次与正则表达式匹配的索引。在这个例子中我们也跟踪调用。执行这个代码的结果会是什么呢：

```
def traceIt[T <: Iterator[_]](it: T) = {
  println(s"TRACE: using iterator '${it}'")
  it
}

val msg = "I love Scala"
println("First match index: " +
  traceIt("a".r.findAllIn(msg)).start)
println("First match index: " +
  "a".r.findAllIn(msg).start)
```

可能的结果

1. 打印出：

```
TRACE: using iterator 'non-emptyiterator'
First match index: 9
First match index: 9
```

2. 两个语句都抛出运行时异常。

3. 第一个语句打印出：

```
TRACE: using iterator 'non-emptyiterator'
First match index: 9
```

第二个语句抛出运行时异常。

4. 第一个语句抛出运行时异常，第二个语句打印出：

```
First match index: 9
```

解释

尽管正则表达式"a"看上去好像应该一定会匹配字符串"I love scala"，你可能奇怪：是什么原因导致没有发现匹配的结果呢？或者你可能会怀疑是打印 trace 方法里的迭代引起了错误吗？要不然，两个语句一定会打印"I love scala"中第一次出现的"a"，它真的在索引 9 的位置吗？（译者注：索引的下标是从 0 开始的）。

并非如此。事实上，正确的答案是 3。

```
scala> println("First match index: " +
         traceIt("a".r.findAllIn(msg)).start)
TRACE: using iterator 'nonempty iterator'
First match index: 9

scala> println("First match index: " +
         "a".r.findAllIn(msg).start)
java.lang.IllegalStateException: No match available
  at java.util.regex.Matcher.end(Matcher.java:389)
  at scala.util.matching.Regex$MatchIterator.end(
      Regex.scala:667)
  ...
```

怎么回事？调试方法并没有“置身事外”？实际上是调试方式使该语句成功了？除简单地打印出迭代的字符串表达式之外，它还做了什么呢？

什么都没做！只是打印，（或者更准确地说）生成了一个有隐含副作用的 MatchIterator 的字符串表达式。这个作用让该语句的调试方式的调用成功了，但“没跟踪”的版本是失败的。

关键点是，作为失败语句指示器的栈跟踪，Scala 的正则支持建立在 Java 的 java.util.regex 包之上。尤其是 Scala 的 MatchIterator 是由 java.util.regex.Matcher 支持的，它具有如下特征[①]：

一个匹配器的显式声明是初始化未定义的；在匹配成功之前试图查询任何部分都会抛出 IllegalStateException 异常。

在你能调用 start，end 或任何其他的代表 Matcher 的 MatchIterator 上的方法之前，首先需要初始化那个 matcher！由于 Iterator 的 toString 实现，这是 MatchIterator 继承的方法叫 hasNext，调用 MatchIterator 的 toString 方法意外地实现了对 matcher 的初始化。这反过来又试图去发现一个匹配并初始化匹配器。

讨论

显然，依赖 Iterator.toString 方法的实现细节并不是一个特别聪明的策略。无论如何，你通常都不会对使用 MatchIterator 时的字符串表达式感兴趣。更可靠的方法是调用 hasNext（或者如果你确定 iterator 是非空的就用 next()）。

```
scala> { // without a block the REPL will call toString
         val mi = "a".r.findAllIn(msg)
         mi.hasNext // initialize the matcher
         println("First match index: " + mi.start)
       }
First match index: 9
```

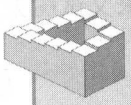
如果你正打算用一个 for 表达式或通过调用 foreach，map 等来对匹配的结果进行迭代，就不需要做任何事情。这些方法调用会为你初始化匹配器。

通过使用 Regex 的 findAllMatchIn 方法能完全避免这个问题。它会将 iterator 转换成 Iterator[Match]，这等同于调用 MatchIterator 的 matchData 方法，这个方法将 iterator 转换成 Iterator[Match]。简单地

① 参见 Javadoc，java.util.regex.Matcher。[Ora]

说就是：这两种情况都是如果不先迭代到一个初始化潜在匹配器的匹配就不能调用像 start 这样“危险的”方法。

```
scala> {
         val mi = "a".r.findAllMatchIn(msg)
         println("First match index: " + mi.next().start)
       }
First match index: 9

scala> {
         val mi = "a".r.findAllIn(msg).matchData
         println("First match index: " + mi.next().start)
       }
First match index: 9
```



优先选择 Regex 的 findAllMatchIn 方法而不是 findAllIn，或者通过调用 MatchIterator.matchData 方法将 findAllIn 返回的 MatchIterator 转换成一个 Iterator[Match]。

第 21 章

填充

Scala 提供了一个增量字符串构建器，它与 java.lang.String Builder 有相同的名字。Scala 的 StringBuilder 与 Java 的大部分相似，主要的不同是提供与 Java 类相同方法的地方会与 Scala 集合库冲突。像 java.lang.StringBuilder 一样，Scala 的 StringBuilder 也不是同步的，留给你用其他方法处理线程安全。

下面的程序展示一个 StringBuilder 的实例。让我们看看它做了什么吧！

```
implicit class Padder(val sb: StringBuilder) extends AnyVal {
  def pad2(width: Int) = {
    1 to width - sb.length foreach { sb += '*' }
    sb
  }
}

// length == 14
val greeting = new StringBuilder("Hello, kitteh!")
println(greeting pad2 20)

// length == 9
val farewell = new StringBuilder("U go now.")
println(farewell pad2 20)
```

可能的结果

1. 打印出：

```
Hello, kitteh!******
U go now.***********
```

2. 第一个语句打印出：

```
Hello, kitteh!*
```

第二个语句抛出异常。

3. 第一个语句抛出异常，第二个语句打印出：

```
U go now.**********
```

4. 两个语句都编译失败。

解释

这个代码看上去足够简单。它由一个用 pad2 方法透明地增强了 StringBuilder 的隐式的值类[①]组成。pad2 方法用特定数目的星号填充一个字符串。接着这个方法被调用两次。

尽管第 1 个回答可能貌似更加合理，但事实上它并不是正确答案。正确的答案是 2：

```
scala> println(greeting pad2 20)
Hello, kitteh!*

scala> println(farewell pad2 20)
java.lang.StringIndexOutOfBoundsException:
    String index out of range: 10
  at j.l.StringBuilder.charAt(StringBuilder.java:55)
  at s.c.m.StringBuilder.apply(StringBuilder.scala:114)
  at s.c.m.StringBuilder.apply(StringBuilder.scala:28)
```

值类

你可能已经注意到 padder 是 AnyVal 的一个子类。这使它是一个值类。值类通过避免对象分配减少了运行时负荷。扩展方法是值类的一个典型的用例[a]，例如 pad2。

[a] Harrah，《值类和普遍特质》。[Har]

```
at s.c.i.Range.foreach(Range.scala:141)
at Padder$.pad2$extension(<console>:9)
...
```

① Suereth，《隐式类》。[Sue]

随着我们逐步解释，真相就在细节中，所以再仔细看看 pad2 方法的实现。表达式 1 to width - sb.length 是一个范围，它的 foreach 方法接受一个函数作为参数：

```
final def foreach(f: (A) => Unit): Unit
```

似乎我们并非正在传递一个函数，因为+=方法（本质上是 StringBuilder.append 方法的一个别名）返回 StringBuilder 本身。代码确实编译了，然而，这只意味着 StringBuilder 本身是一个函数。看看 Scaladoc 就能确认 Scala 的 StringBuilder 是从 Function1 继承来的，它将你带到 StringBuilder 的 apply 方法：

```
def apply(index: Int): Char
```

这等同于 charAt。

返回结果：这个可增长的集合在索引 idx 位置的元素，0 表示第一个元素。

现在事情开始合理了。每次 foreach 迭代的时候不是执行表达式 sb += '*'，而是一评估 foreach 的参数就调用+=。事实上每次调用的是 StingBuilderapply 的 apply 方法，它在给定的索引位置检索字符，但并不将它赋值给别的变量。方法 pad2 实际上等同于下面的代码：

```
def pad2(width: Int) = {
  val appendedSb = sb += '*'
  // apply calls charAt
  1 to width - sb.length foreach appendedSb.apply
  sb
}
```

这两种情况传递给 pad2 的宽度都是 20，运行"Hello, Kitteh!"字符串没有错误，因为传递给 charAt 的最大索引是 6。StringBuilder 的值（输入字符串加一个填充星号）就打印出来。第二个字符串"U go now"更短，所以范围扩展到 11，这比"U go now."字符串的长度要长。方法 charAt 最终用索引 10 被调用，这导致 tringIndexOutOfBoundsException 的结果。

讨论

为了正确地填充字符串，你所要做的就是显式地指定函数迭代。

```
def pad2(width: Int) = {
  1 to width - sb.length foreach { _ => sb += '*' }
```

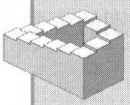
```
  sb
}
```

如果你用了 for 表达式这会自动发生，编译器会在后台将 for 表达式翻译成一个 foreach 调用。

```
def pad2(width: Int) = {
  for (_ <- 1 to width - sb.length) { sb += '*' }
  sb
}
```

你如果能在 StringBuilder 上使用已有的 padTo 方法就更好了：

```
scala> println(new StringBuilder("Hello, kitteh!")
         .padTo(20, '*').mkString)
Hello, kitteh!******
scala> println(new StringBuilder("U go now.")
         .padTo(20, '*').mkString)
U go now.***********
```



在给返回函数的 foreach 传递表达式时一定要小心。一般说来，在写你自己的实用工具代码之前，先检查 Scala 集合库中是否已经存在合适的方法。

第 22 章

投影

尽管 Scala 中所有的值都是对象，但它的基本值类型（Byte，Short，Int 等）在尽可能的地方都会被编译成 Java 中的原始相应部分。这就可以把这些 Scala 类型的实例当成常规对象对待，从而简化了编程模式。反过来也可以将 Java 原始部分当成 Scala 值类型对待，也使得用 Java 库变得更加容易了。

然而，在 Java 和 Scala 集合类型间却没有类似的映射（必须由程序员来实现在它们之间的转换）。Scala 有 JavaConversions 和 JavaConverters 两个对象可以处理这种转换。JavaConverters 通常是首选，因为它使转换在代码中更明显。

下面的程序演示如何在 Scala 中使用 Java 集合。让我们看看这个程序会做什么吧！

```
import collection.JavaConverters._

def javaMap: java.util.Map[String, java.lang.Integer] = {
  val map =
    new java.util.HashMap[String, java.lang.Integer]()
  map.put("key", null)
  map
}

val scalaMap = javaMap.asScala
val scalaTypesMap =
  scalaMap.asInstanceOf[scala.collection.Map[String, Int]]

println(scalaTypesMap("key") == null)
println(scalaTypesMap("key") == 0)
```

可能的结果

1. 打印出：

```
true
true
```

2. 两个 println 语句都抛出一个空指针例外 NullPointerException。

3. 打印出：

```
true
false
```

4. 打印出：

```
false
true
```

解释

第一步，让我们先浏览程序。如它的名字所暗示的，javaMap 方法会模仿调用 Java 库。它返回一个有着 java 键和值（key-value）类型的 Java map。

asScala 方法将 Java map 转换成 Scala map：

```
scala> javaMap
res0: java.util.Map[String,Integer] = {key=null}

scala> val scalaMap = javaMap.asScala
scalaMap: scala.collection.mutable.Map[String,Integer] =
  Map(key >null)
```

此时，你有一个 Scala map，但是以一名 Scala 程序员的视角看，值类型 java.lang.Integer 看起来却有些不同。为了将它转换到 Scala 类型，需要将 java.lang.Integer 投影到 Scala.Int：

```
scala> val scalaTypesMap = scalaMap.asInstanceOf[
         scala.collection.Map[String, Int]]
scalaTypesMap: scala.collection.Map[String,Int] =
  Map(key ->null)
```

最终得到一个 Scala 的 map，现在就可以开始用了（在这个例子中随后的两个 println 语句）。

你可能已经注意到最后那段代码的结果透露出 scalaTypesMap 的实际值是 Map(key ->null)。你可能因此认为第一个 println 语句一定是 true，第二个是 false（如候选答案是 3）。

在 REPL 中执行的结果却并非如此：

```
scala> println(scalaTypesMap("key") == null)
true

scala> println(scalaTypesMap("key") == 0)
true
```

所以正确的答案是 1。一个值怎么可能同时等于 null 和 0 呢？实际上，是两方面的原因：一是 scalaMap 的投影；二是 java.lang.Integer 和 scala.Int 两者并不完全相同。

你也许已经注意到，即便 scalaTypesMap 的值类型是 Int，但它还是含有值 null。值类型 Int 从定义上绝不是 null，因为 Int 从 AnyVal[①]继承。然而，这里 null 位于集合内。这之所以重要，是因为 Scala 集合像 Java 集合（除 arrays 以外）一样不能直接存 Java 原始类型。因为集合是通用的，而通用类服从于类型擦除，集合类型的元素被擦除到 AnyRef(java.lang.Object)。

所有的 Scala 值类型包括 Int 都是 AnyVal，因而将它们存在集合中时需要装箱进 AnyRef 打包器类型中[②]。所以每个 Scala Map[String,Int]实际上都含有内部的 Integer。因而编译器一直是从 Map[String,Int]读出 Integer 值。

这个谜题真正的行动是两个 println 语句。第一个 println 语句将 map 值与 null 比较：

```
println(scalaTypesMap("key") == null)
```

因为从 map 抽取出来的值是 AnyRef 打包器类型，它直接与 null（也是一个 AnyRef）比较。而 null == null 的结果是 true，这没什么可惊讶的。

另一方面，第二个 println 语句将 map 值与 0 比较：

```
println(scalaTypesMap("key") == 0)
```

由于性能原因，编译器总是试图尽可能在原始类型而不是打包器类型上进行操作，这意味着编译器拆箱从 map 抽取出打包器类型，其结果代码类似于：

```
println(unbox(scalaTypesMap("key")) == 0)
```

如果反编译代码，就能看到抽取过程调用编译器用于执行拆箱：

---

① 有关值类型更详细的讨论，请参加第 28 章。

② Odersky，Spoon，Venners，《Scala 编程》。[Ode10]

```
Predef$.MODULE$.println(BoxesRunTime.boxToBoolean(
  scalaTypesMap.apply("key") == null));
Predef$.MODULE$.println(BoxesRunTime.boxToBoolean(
  BoxesRunTime.unboxToInt(scalaTypesMap.apply("key")) == 0));
```

BoxesRunTime.unboxToInt 方法执行如下[①]：

```
public static int unboxToInt(Object i) {
  return i == null ? 0 : ((java.lang.Integer)i).intValue();
}
```

这就是为什么当 scalaTypesMap("key")的结果为第二个 println 语句拆箱时会返回 0。结果就是这个相等比较也被评估为 true。

通过检查编译器某个阶段的输出，就能观察到这个行为。这里最关键的是消除阶段。给编译器传递-Xprint:erasure 参数就能打印出下面的信息（为了清晰进行了简化）：

投影 Null

Scala 语言规范说：在 null（它有 null 类型）上调用 asInstanceOf[T]返回类型 T[②]的缺省值。

例如，由于 Int 的缺省值是 0，表达式 null.asInstanceOf[Int]也评估为 0。

```
def javaMap(): java.util.Map = {
  val map: java.util.HashMap = new java.util.HashMap();
  map.put("key", null);
  map
};
// types erased
val scalaMap: collection.mutable.Map =
  mapAsScalaMapConverter(javaMap()).asScala()
  .asInstanceOf[collection.mutable.Map]();
val scalaTypesMap: collection.Map =
  scalaMap.asInstanceOf[collection.Map]();
println(scala.Boolean.box(
```

① BoxesRunTime.unboxToInt 的逻辑与 Predef.Integer2int 和 scala.Int.unbox 不同，传给它 null 就会抛出 NullPointerException 异常。

② Odersky，《Scala 语言规范》，6.3 节。[Ode14]

```
  scalaTypesMap.apply("key").==(null)));
// wrapper type unboxed to primitive
println(scala.Boolean.box(
  unbox(scalaTypesMap.apply("key")).==(0)))
```

讨论

你可能会问，如果对返回原始的 Java key-value 类型的 scalaMap 用相同的 println 语句会是什么结果呢？

```
println(scalaMap("key") == null)
println(scalaMap("key") == 0)
```

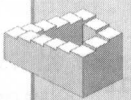
注意消除阶段后的差别：

```
println(scala.Boolean.box(scalaMap.apply("key").==(null)));
println(scala.Boolean.box(
  scalaMap.apply("key").==(scala.Int.box(0))))
```

关键差别在最后一行。不是拆箱就有了 java.lang.Integer 类型的 map 值，而是编译器装箱了字面量 0 并与结果 Anyrefs 进行比较。这时结果就与期望更加一致了。

```
scala> println(scalaMap("key") == null)
true

scala> println(scalaMap("key") == 0)
false
```

简而言之，当存在于集合里的打包的值类型与另一个值类型比较时会被拆箱。相比之下，如果声明的集合类型是一个打包器类型就不会发生拆箱，而是值类型会被装箱。



投影天生就是不安全的，一般应该避免，因为它完全绕开了类型检查。尤其是将 Scala 值投影到 Java 打包器类型时，反过来从 Java 投影到 Scala 也一样，都会产生意想不到的结果。要让编译器隐式地处理转换而不是显式地自己做投影。

第 23 章

构造器参数

Scala 支持两种传递参数的方式：

1. 按值传递，这种方式会对参数传递给方法之前对它进行评估。这是缺省的方式。

2. 按名字传递，这种方式是只有当在方法内引用参数时才对参数进行评估[①]。按名字传递参数要用=>符号事先设定。

按名字传递参数适用于希望避免在方法调用之前评估参数的情况，尤其是如果评估成本较高时。不过与 lazy.val 不同的是，按名字传递参数每次在方法内引用这些参数时都要进行评估：

```
def mod(a: => Double) = if (a >= 0) a else -a

scala> mod({ println("evaluating"); -5.2 })
evaluating
evaluating
res0: Double = 5.2
```

Scala 另外一个很好的特性是在传递一个块时可以省略圆括号。这能让方法调用看上去和感觉上都像一个内建的控制结构[②]：

```
List(1, 2, 3) foreach { e => println(math.abs(e)) }
```

下面的程序结合了这两个特征。让我们看看它会做什么吧。

```
class Printer(prompter: => Unit) {
  def print(message: String, prompted: Boolean = false) {
    if (prompted) prompter
    println(message)
  }
```

① Odersky，《Scala 语言规范》，4.6.1 节。[Ode14]

② 谜题 35 的特色也是方法调用中的大括号和圆括号。

```
}

def prompt() {
  print("puzzler$ ")
}

new Printer { prompt } print (message = "Puzzled yet?")
new Printer { prompt } print (message = "Puzzled yet?",
  prompted = true)
```

可能的结果

1. 打印出：

```
Puzzled yet?
puzzler$ Puzzled yet?
```

2. 打印出：

```
puzzler$ Puzzled yet?
puzzler$ Puzzled yet?
```

3. 打印出：

```
puzzler$ Puzzled yet?
Puzzled yet?
```

4. 编译失败。

解释

这两个 `Printer.print` 调用似乎都遵从相同的代码路径：

1. 创建一个新的 `Printer` 实例，用 `prompt` 作为 `prompter` 的构造器参数 `prompter`。

2. 给 `Prompter` 的参数是按名字传递的，所以先不进行评估且什么都没打印到控制台。

3. 调用 `print` 方法。第一次调用没有给 `prompter` 提供值，所以它是缺省值 `false`。因而 `prompter` 就不会被调用，只打印出 `message` 的值。第二次调用 `print`，`prompted` 显式设置为 `true`，致使 `prompter` 在 `message` 输出之前就被调用。

所以第一个候选答案似乎是正确的。可惜你从 REPL 运行代码就会发现这个分析不正确。正确的答案是 2！

```
scala> new Printer { prompt } print (message =
         "Puzzled yet?")
puzzler$ Puzzled yet?

scala> new Printer { prompt } print (message =
         "Puzzled yet?", prompted = true)
puzzler$ Puzzled yet?
```

两次调用 print 输出结果都相同的事实相当有趣。它证明以上分析漏掉了某些关键的与构造器参数传递的方式有关的内容。尤其是，大括号只有在方法参数的情况才可以代替圆括号。另一方面，构造器参数总是需要在圆括号里提供。

简而言之，下面的表达式并不相等：

```
new Printer(prompt)
new Printer { prompt }
```

第一个语句创建一个 Printer 实例，用 prompt 作为构造器参数。第二个语句则完全不同：它用 no-arg 实例化一个匿名子类，主构造器。所以，prompt 不是做为构造器参数，而是作为匿名子类的构造器的一部分被调用。

尽管如此，Printer 有一个类参数（primpter），而且它看上去并不像创建新实例时所提供的值。如果 prompt 不是构造器参数，代码是怎么编译的呢，将 Printer 声明看作一个类参数吗？

首先，可以在 Scala 语言规范里找到一个解释，它说明如果没有给出显式的构造器参数，就提供一个空的参数列表 ()[①]。所以第一次调用 print 实际上看上去是这样的：

```
new Printer() { prompt } print (message = "Puzzled yet?")
```

但是这里并没有提供一个构造器参数，所以它仍不清楚代码如何编译。这就该另一个语言特性——参数适配器起作用了：编译器试图通过增加 Unit 值 ()“修复”参数列表中丢失的参数，并看看是否能进行结果类型检查[②]。于是就产生了下面的表达式：

```
new Printer(()) { prompt } print (message = "Puzzled yet?")
```

---

① Odersky，《Scala 语言规范》，5.1.1 节。[Ode14]

② 有关参数适配器的更详细的讨论，请参见第 32 章。

现在你终于看到了这个谜题全部。因为 prompt 方法是类定义的一部分（准确地说是 no-arg 主构造器），两种情况都是 Printer 一被实例化就执行 prompt。当 prompt 被调用的时候，给 prompter 传递的参数值 Unit() 什么都没做。因此，两次调用 print 的结果是一样的。

讨论

用-Xlint 参数在 REPL 中运行的结果是有一个警告[①]：

```
scala> new Printer { prompt } print (message =
         "Puzzled yet?")
<console>:10: warning: Adapting argument list
    by inserting (): this is unlikely to be what you want.
```

Scala 代码静态分析

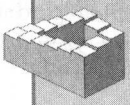
Scala 编译器的-Xlint 参数开启了附加的编译器警告，这个警告可以标识出使用了可疑的语言，包括适配参数列表。

```
        signature: Printer(prompter: => Unit): Printer
  given arguments: <none>
after adaptation: new Printer((): Unit)
   new Printer { prompt } print (message = "Puzzled yet?")
       ^
puzzler$ Puzzled yet?
```

为了得到期望的结果，你要做的就是使用圆括号以确保将 prompt 传递为一个构造器参数：

```
scala> new Printer(prompt) print (message = "Puzzled yet?")
Puzzled yet?

scala> new Printer(prompt) print (message = "Puzzled yet?",
         prompted = true)
puzzler$ Puzzled yet?
```

① 这个警告是由 Scala2.11 缺省给出的，Scala2.11 不支持参数适配器。



记住，要将构造器参数放在圆括号里。只有当用作方法参数的时候才能用大括号代替圆括号。

第 24 章

Double.NaN

浮点数和整数运算的一个差别是存在特别的浮点值 NaN。一般情况下，这个值表现的是可预见的，但是你仍需要知道它能如何起作用而影响你的代码。

这段代码是对两个浮点值集合进行排序。让我们看看执行下面的代码结果是什么吧！

```
def printSorted(a: Array[Double]) {
  util.Sorting.stableSort(a)
  println(a.mkString(" "))
}

printSorted(Array(7.89, Double.NaN, 1.23, 4.56))
printSorted(Array(7.89, 1.23, Double.NaN, 4.56))
```

可能的结果

1. 打印出：

```
1.23 4.56 7.89 NaN
1.23 4.56 7.89 NaN
```

2. 打印出：

```
1.23 4.56 7.89 NaN
1.23 7.89 NaN 4.56
```

3. 打印出：

```
NaN 1.23 4.56 7.89
NaN 1.23 4.56 7.89
```

4. 打印出：

```
1.23 4.56 7.89 NaN
1.23 4.56 NaN 7.89
```

解释

你可能要问 Double.NaN 是否会被认为比所有其他的 Double 大或小呢？或者有 NaN 参与的排序不会出错吧？你确定 NaN 值的出现不会导致 array 的排序不正确吗？

噢，是的，它能（有时候）。正确的答案是 2：

```
scala> printSorted(Array(7.89, Double.NaN, 1.23, 4.56))
1.23 4.56 7.89 NaN

scala> printSorted(Array(7.89, 1.23, Double.NaN, 4.56))
1.23 7.89 NaN 4.56
```

为了理解这里发生了什么，你需要首先考虑 NaN 的几个属性。为遵从 IEEE 754[①]，在 Java 和 Scala 中对涉及 NaN 参与的用==，<，>，<=和>=的比较，结果总是 false；涉及 NaN 用！=的比较，结果总是 true。

```
scala> 1.0 < Double.NaN
res0: Boolean = false

scala> 1.0 > Double.NaN
res1: Boolean = false

scala> Double.NaN != Double.NaN
res2: Boolean = true
```

在 Scala 2.10.0[②]中，util.Sorting 用的是 Scala 缺省隐含的对浮点数的排序方法 Ordering，它遵从 IEEE 754。结果在浮点数上用 Ordering 排序是不一致的。例如，compare(1.0, Double.NaN) 是负的，暗示着 1.0 小于 Double.NaN。另一方面，直接比较 lt(1.0, Double.NaN) 返回 false，与 IEEE 标准一致。这又暗示了 1.0 并不是小于 Double.NaN：

```
val doubleOps = implicitly[Ordering[Double]]
```

① IEEE 754 是浮点运算标准。

② 参见 SI-5104，“(Double.NaN min 0.0) yields 0.0，应该是 NaN。”[Dou]

```
scala> doubleOps.compare(1.0, Double.NaN)
res12: Int = 1

scala> doubleOps.lt(1.0, Double.NaN)
res13: Boolean = false
```

这个不一致是导致本代码例子观察到的行为的根本原因。为了看看它是如何引起代码例子中的第二个数组，`Array(7.89,1.23,Double.NaN,4.56)` 不正确排序的，你需要再仔细地看看 `Sorting.stableSort` 的实现，这是 Scala 缺省稳定的排序算法，是对几乎所有 array 和 sequence 预定的排序算法[①]。

为了排序一个数组，`stableSort` 首先从中间将它分开，通过缺省使用 `Ordering.it` 比较元素来递归地对的子数组排序。为了产生最后的结果，需要合并两个排序后的子数组：只要第一个子数组中的元素小于或等于第二个子数组的当前“引用元素”就从第一个子数组中取出元素，再次使用 `Ordering.it` 比较元素，只有当引用元素（初始化的，第一个元素）较大的时候才把它加到结果数组中，于是第二个数组的下一个元素就变成了新的引用。

这个完整的过程关键依赖第二个子数组被认为是已经排序过的事实：在这种情况下，如果第一半的当前元素小于或等于第二半的参考元素，它就必然也小于或等于那半里所有随后的元素。本例的代码中的第二个数组会发生什么呢？首先 `stableSort` 递归地排序子数组 `1.23`，`7.89` 和 `Double.NaN`，`4.56`，结果排序好的子数组是 `1.23`，`7.89`，`Double.NaN`，`4.56`。根据 IEEE 规格，第二个子数组的确是被正确地排序了，因为 `Ordering.lt(4.56, Double.NaN)` 是 `false`，如 `Double.NaN <= 4.56`。

现在，算法开始合并过程，将第一半的当前元素 `1.23` 与第二半的引用元素 `Double.NaN` 比较。因为 `1.23` 小于或等于 `Double.NaN`（根据 `Ordering.lt`），将它增加到结果数组。接着 `stableSort` 将第一半的下一个元素 `7.89` 再与 `Double.NaN` 比较，再次得到结果，因为它小于或等于 `Double.NaN`，所以应该被加到结果数组。

错误正在于此：当 `Ordering.lt(Double.NaN, 7.89)` 时，例如 `7.89 <= Double.NaN` 是 `true`，结论是 `7.89` 小于第二个子数组中的所有随后的元素——`4.65`，在这个例子中是 `false`。然而此时，第一个子数组的两个元素都已经处理过了。算法就得到结论第二个子数组中剩下的元素都一定比已经加到结果数据中的值要大，所以就简单地把它们都加入到结果里。

---

① 如果一个排序算法能保持相等元素的排序是不变的，那么这个排序算法就是稳定的。

讨论

特殊的 Double.NaN 值的出现至少给你一个提醒：什么可能会引起意想不到的行为。如果有任何正在执行的“数据清洗”过程时产生了这样的值，要想弄明白怎么回事就更加困难了。假设正在试图调试如下代码：

```
import util.Sorting.stableSort

def filterNaN(arr: Array[Double]) = arr filter { !_.isNaN() }

val filterBeforeSort =
  filterNaN(Array(1.23, 7.89, Double.NaN, 4.56))

stableSort(filterBeforeSort)

scala> println(filterBeforeSort mkString " ")
1.23 4.56 7.89

val sortBeforeFilter = Array(1.23, 7.89, Double.NaN, 4.56)
stableSort(sortBeforeFilter)

scala> println(filterNaN(sortBeforeFilter) mkString " ")
1.23 7.89 4.56
```

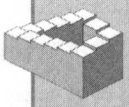
这里，真的看不出什么问题——你只是简单地得到一个不正确的排序数组。

为了避免这个问题，可以为 Double 定义一个 ordering（或者给排序算法传递一个显式的比较函数），使它能像 java.lang.Double.compareTo 一样一致地处理 NaN.Double.compare，把 Double.NaN 当作与它自己相等或比所有其他的 double 值都大的数，这是个好方法：

```
def printSortedByDoubleCompare(a: Array[Double]) {
  def ltWithNaNGtAll(x: Double, y: Double) =
    (x compare y) < 0
  util.Sorting.stableSort(a, ltWithNaNGtAll _)
  println(a.mkString(" "))
}

scala> printSortedByDoubleCompare(
    Array(7.89, Double.NaN, 1.23, 4.56))
1.23 4.56 7.89 NaN
```

```
scala> printSortedByDoubleCompare(
    Array(7.89, 1.23, Double.NaN, 4.56))
1.23 4.56 7.89 NaN
```

注意：Double.compare 的行为并不遵从 IEEE 754。令人惊讶的是，至今还没有一种符合 IEEE 规格的排序方式能一致地处理含有 NaN 的浮点数。



至于 Scala 2.10.0，用缺省的 ordering 排序浮点数不会正确地处理 NaN。当排序可能含有 NaN 的浮点集合时，需要定义你自己的 Ordering 或者为排序算法提供显式的比较函数。

第 25 章

getOrElse

Scala 集合库中的许多方法都有一个 option 返回类型，要防止它们可能会传递值失败。从 option 中抽取值的推荐的方式是通过 getOrElse 方法，这个方法让你能在 Option 为空时提供一个缺省值。

下面这段程序演示一个使用 getOrElse 的例子。让我们看看它会做什么吧！

```
val zippedLists = (List(1,3,5), List(2,4,6)).zipped
val (x, y) = zippedLists.find(_._1 > 10).getOrElse(10)

println(s"Found $x")
```

可能的结果

1. 打印出：

```
Found 10
```

2. 打印出：

```
Found ()
```

3. 编译失败。

4. 抛出运行时异常。

解释

在你试图理解这里发生了什么之前，让我们先通过一个简短的例子快速回顾一下 zipped 的行为会很有帮助：

```
scala> val zippedLists = (List(1,3,5), List(2,4,6)).zipped
zippedLists: scala.runtime.Tuple2Zipped[Int,List[Int],
  Int,List[Int]] = scala.runtime.Tuple2Zipped@3d38da0d

scala> zippedLists.toList // force evaluation
res0: List[(Int, Int)] = List((1,2), (3,4), (5,6))
```

总之，zipped 取出两个列表中匹配的元素将它们组合在一起成为元组，结果返回一系列这样的元组。

既然你已经确定 zipped 能做什么了，现在让我们回过头来看看这个谜题。Find 方法搜索匹配断言的第一个元组，在这个例子中，是第一个元素大于 10 的元组。如果返回 None，就返回 getOrElse 方法的缺省值。注意，缺省值类型 Int 与 find 正在检查的元素类型(Int, Int)并不匹配。因为必须要将表达式的结果分配给一个有两个元素的元组，所以代码会编译失败，对吗？换句话说，第 3 个候选项一定是正确的。

让我们在 REPL 里执行看看：

```
scala> val zippedLists = (List(1,3,5), List(2,4,6)).zipped
zippedLists: scala.runtime.Tuple2Zipped[Int,List[Int],
  Int,List[Int]] = scala.runtime.Tuple2Zipped@3d38da0d

scala> val (x, y) = zippedLists.find(_._1 > 10).getOrElse(10)
scala.MatchError: 10 (of class java.lang.Integer)
```

结果编译倒是成功了，但代码却抛出运行时异常！正确的答案是 4。

这里我们忽略了一点：在 Option[A]的实例上调用 getOrElse 并不一定返回值类型 A：

```
final def getOrElse[B >: A](default: => B): B
```

Option.getOrElse 并不是这样拓宽行为的唯一方式。如 Option.OrElse，Try.recover 和 Future.recover 方法也能返回一个比原始的元素类型更宽的类型。

在这段代码中，推断出的返回类型是 Any（Tuple2 和 Int 的最特定的子类型）。

```
scala> zippedLists.find(_._1 > 10).getOrElse(10)
res0: Any = 10
```

为什么编译器没有抱怨类型不匹配呢？毕竟，Any 值类型是不能给 Tuple2 类型的 val 赋值的。实际上这段代码并没有这样做，而是用元组表示一个模式定义[①②]：

```
val (x, y) = zippedLists.find(_._1 > 10).getOrElse(10)
```

这个表达式展开后如下所示：

```
val a$ = zippedLists.find(_._1 > 10).getOrElse(10) match {
  case (b, c) => (b, c)
}
val x = a$._1
val y = a$._2
```

换句话说，代码编译没有问题，但运行时却由于模式匹配失败而抛出异常。

元组 val 和元组模式

注意，元组类型的 val 定义和正参与元组模式的定义之间在语法上略微不同：

```
val tup: (Int, Int) = ... // regular
val (x: Int, y: Int) = ... // pattern
```

讨论

为了防止编译器推断出比本意更宽的类型，你可以显式地规定期望的类型：

```
scala> val (x, y): (Int, Int) =
         zippedLists.find(_._1 > 10).getOrElse(10)
<console>:9: error: type mismatch;
found    : Int(10)
required : (Int, Int)
       val (x, y): (Int, Int) =
         zippedLists.find(_._1 > 10).getOrElse(10)
                                                ^
```

在类型不匹配不太明显的情况加宽类型可能会在运行时遇到问题。例如，看看下面这段代码，我们没有注意到 Int 和 String 的顺序不正确：

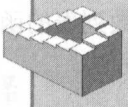
---

① Odersky，《Scala 语言规范》，4.1 节。[Ode14]

② 有关模式定义的详细讨论，请参见第 2 章。

```
def howToPronounce(numAndName: Option[(Int, String)]) = {
  val (num, name) = numAndName.getOrElse(("eight", 8))
  println(s"The word for $num is '$name'")
}

scala> howToPronounce(Some((7, "seven")))
The word for 7 is 'seven'

scala> howToPronounce(None)
The word for eight is '8'
```

这并不是你期望的结果。编译器推断（Any，Any）是 getOrElse 的返回类型，而 num 和 name 都是 Any 类型。为了避免运行时的这种意外，一定要确保你对程序中使用撤退值的代码路径也做了测试。



许多有撤退值的方法能返回一个比原计划更宽的类型。鉴于此，要为这些方法的非平凡表达式指定期望的返回类型。

第 26 章

Any Args

Scala 允许函数的定义有多个参数或多个参数列表（例如，克里化方法）。这让你定义函数时有极大自由，但也意味着如果要改变一个函数的“参数样式”就要重新定义函数。

请看下面这个例子，它是将一个一开始有两个参数的单一参数列表的函数重构为使用克里化参数的函数，然后用不变的方式调用重构前后的方法。

执行下面的代码会是什么结果呢？

```
def prependIfLong(candidate: Any, elems: Any*): Seq[Any] = {
  if (candidate.toString.length > 1)
    candidate +: elems
  else
    elems
}
println(prependIfLong("I", "love", "Scala")(0))

def prependIfLongRefac(candidate: Any)(elems: Any*):
    Seq[Any] = {
  if (candidate.toString.length > 1)
    candidate +: elems
  else
    elems
}
// invoked unchanged
println(prependIfLongRefac("I", "love", "Scala")(0))
```

可能的结果

1. 打印出：

```
love
love
```

2. 打印出：

```
love
ArrayBuffer((I,love,Scala), 0)
```

3. 打印出：

```
love
IloveScala
```

4. 第一个 println 打印出：

```
love
```

第二个语句编译失败。

解释

你可能会问编译器是否能应用某种“兼容性转换”用统一的调用方式去适应新的克里化声明呢。如果不能，第二个 prependIfLongRefac 调用就一定会编译失败吗？

并非如此：

```
scala> println(prependIfLong("I", "love", "Scala")(0))
love

scala> println(prependIfLongRefac("I", "love", "Scala")(0))
ArrayBuffer((I,love,Scala), 0)
```

因此，正确的答案是 2。首先，让我们简单浏览重构前的调用，它表现得与期望一样。这里，第一个参数"I"绑定到 prependIfLong 的候选参数，"love"和"Scala"被绑定到第二个 vararg 参数 elems。“I”不再是一个字符，所以 prependIfLong 返回 vararg 序列 WrappedArray("love", "Scala")没变。它的第一个元素"love"就被打印出来：

```
val prepended = prependIfLong("I", "love", "Scala")

scala> println(prepended(0))
love
```

到目前为止还很直观。在第二个调用重构之后，此时开始变得有趣了。最让

人吃惊的是调用语句没变却居然编译成功了。Println 语句产生了一个相当出乎意料的输出：你原本是希望打印返回序列的第一个元素的。

编译器是如何编译这次调用的呢？重构之后，prependIfLongRefac 期望一个有唯一参数的初始化参数列表。然而，为了与 prependIfLong 的原始的多参数声明匹配还是传递了三个参数。

这里有一个语言规范中没有记录的特性在起作用。在编译器试图寻找能用三个参数的 rependIfLongRefac 版本不成功之后，它就尝试最后一个选择：把所有的参数放进元组并试图将函数应用到这个元组。在这个例子里，自动生成元组实际上成功地将 candidate 绑定到三元组（"I", "love", "Scala"）。

此时，给 prependIfLongRefac 提供的所有参数都用尽了，那么第二个 vararg 参数 elems 怎么办呢？这里，编译器简单地用（0）作为第二个参数，这应该是从方法结果中抽取的第一个元素！因而打印出原打算给 elems 的由 candidate 组成（因为 candidate.toString 一定比一个字符长）的结果。

换句话说，第二个调用等于：

```
val prepended = prependIfLongRefac(("I", "love", "Scala"))(0)

scala> println(prepended)
ArrayBuffer((I,love,Scala), 0)
```

讨论

在自动创建元组[①]时，你可以分别用-Ywarn-adapted-args 和-Yno-adapted-args 编译器参数来发出警告和失败：

-Ywarn-adapted-args 如果将参数列表修改成与接受者匹配就会发出警告。

```
scala> :settings +Ywarn-adapted-args

scala> println(prependIfLongRefac("I", "love", "Scala") _)
<console>:9: warning: Adapting argument list by creating
  a 3tuple:this may not be what you want.
       signature: prependIfLongRefac(candidate: Any)
```

① 不幸的是，如果适配器没有出现在参数列表中，就不会触发警告和失败。

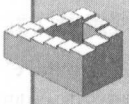
```
         (elems: Any*): Seq[Any]
  given arguments: "I", "love", "Scala"
after adaptation: prependIfLongRefac(("I", "love", "Scala"):
  (String, String, String))
            println(prependIfLongRefac(
                                       ^
<function1>
```

-Yno-adapted-args 则不需要参数列表与接受者匹配。

```
scala> :settings +Yno-adapted-args

scala> println(prependIfLongRefac("I", "love", "Scala") _)
<console>:9: error: too many arguments for method
  prependIfLongRefac: (candidate: Any)(elems: Any*)Seq[Any]
              println(prependIfLongRefac(
                                         ^
```

指定第一个参数作为命名参数是另外一种阻止编译器默认自动创建元组的方式：

```
scala> println(prependIfLong(
         candidate = "I", "love", "Scala")(0))
love

scala> println(prependIfLongRefac(
         candidate = "I", "love", "Scala")(0))
<console>:9: error: too many arguments for method
  prependIfLongRefac: (candidate: Any)(elems: Any*)Seq[Any]
              println(prependIfLongRefac(
                                         ^
```



当用 Any 参数类型定义方法或函数时，尤其是重构这样的方法时，一定要小心自动创建元组的可能。你还可以用编译器参数-Ywarn-adapted-args 和-Yno-adapted-args 来警告或阻止自动创建元组。

第 27 章

null

Scala 的一个强大之处是它与 Java 可以互操作，这就可以没有任何性能损失地重用 Java 库和工具。你可以从 Scala 访问并继承 Java 类、调用 Java 方法等。

不幸的是，Java 方法经常返回 null，这通常说明没有数据可用。尽管这是 Java 语言的惯用法，但是模糊的 null 还是会带来问题。无论返回的 null 表示的是未初始化、不存在或是空值往往都是模糊的。此外，方法的调用者也不清楚他们调用的方法会返回 null，结果，就有可能忘记显式地检查 null。这就会引起超出意外的大量 bug。

下面的程序是一个从 Java 方法返回空值的例子。为简明起见，这段代码只是模拟调用一个 Java 方法显式地返回 null（并没有真的调用 Java 方法）。让我们看看这个程序会做什么吧！

```
def objFromJava: Object = "string"
def stringFromJava: String = null

def printLengthIfString(a: AnyRef): Unit = a match {
  case str: String =>
      println(s"String of length ${str.length}")
  case _ => println("Not a string")
}

printLengthIfString(objFromJava)
printLengthIfString(stringFromJava)
```

可能的结果

1. 打印出：

```
Not a string
String of length 0
```

2. 第一个调用 printLengthIfString 打印出：

```
Not a string
```

第二个抛出 NullPointerException。

3. 打印出：

```
String of length 6
Not a string
```

4. 第一个调用 printLengthIfString 打印出：

```
String of length 6
```

第二个抛出 NullPointerException。

解释

正确的答案是 3：

```
scala> printLengthIfString(objFromJava)
String of length 6

scala> printLengthIfString(stringFromJava)
Not a string
```

为了理解其原因，让我们先从 Scala 的观点看看 null 字面量。如图 27.1 所示是 Scala 的类层次图。

我们注意到，null 在层次图的底部，在 Nothing 上面。Null 是 Scala 中仅仅为了与 Java 兼容而存在的特殊类型。它只有一个实例：null。类型 Null 是所有引用类型的子类型，因此，可以将 null 分配给一个变量或传递为任何引用类型的参数：

```
scala> val s: String = null
s: String = null

scala> def say(s: String) = println(s)
say: (s: String)Unit

scala> say(null)
null
```

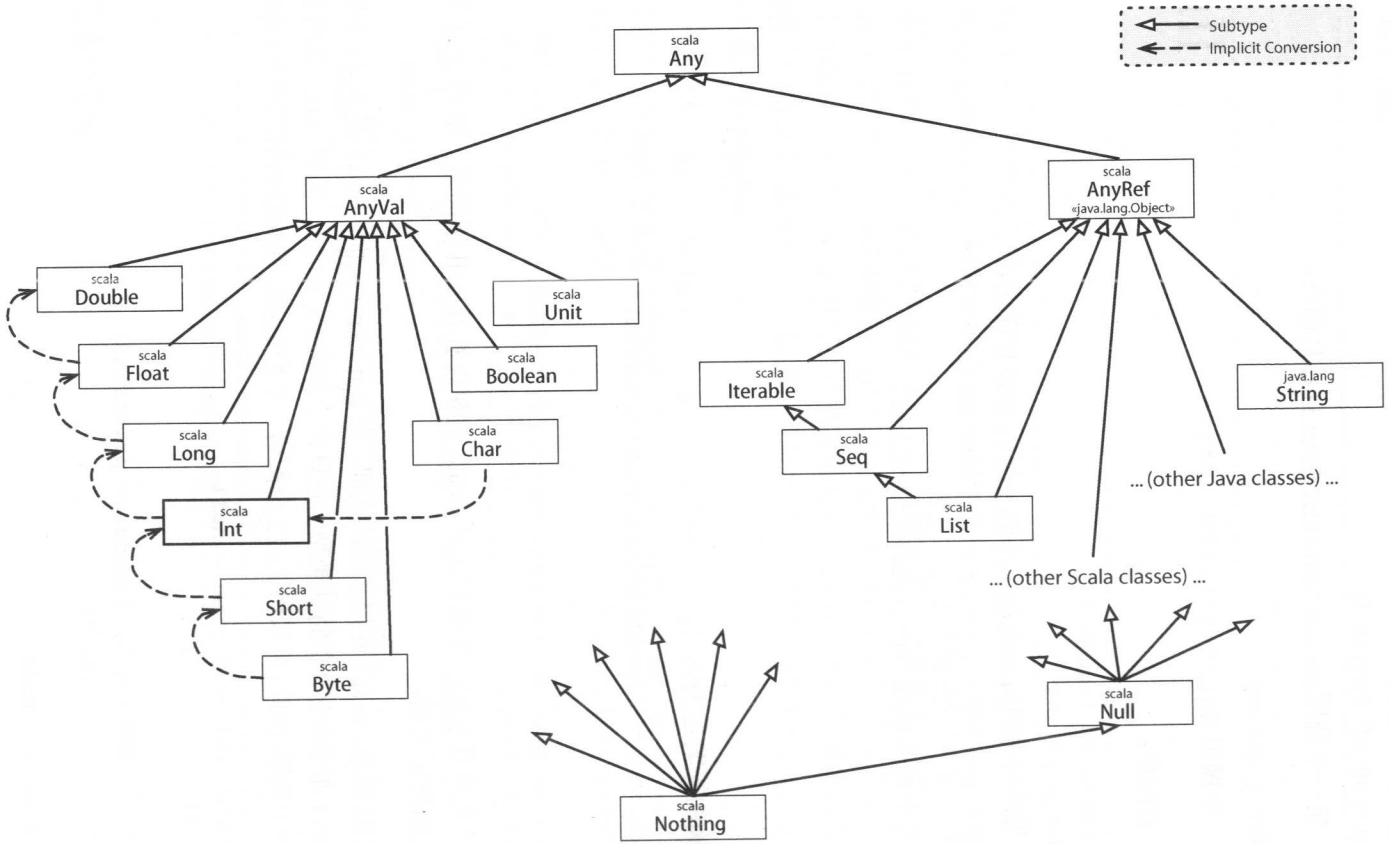


图 27-1　Scala 的类层次

然而，null 除了 Null 以外并不是任何其他类型的实例：

```
scala> val s: String = null
s: String = null

scala> s.isInstanceOf[String]
res0: Boolean = false
```

Scala 从 Java 继承[①]的这个行为是相关的，其原因是这段代码的关键部分是 printLengthIfString 方法中的模式匹配。在 Scala 中，类型的模式匹配通过 isInstanceOf 方法来实现，它基于运行时类型执行检查。所以，第一次调用方法 printLengthIfString 匹配了第一个 case 语句，第二次调用，即使 stringFromJava 值的编译时类型是 String，缺省情况下还是失败了，这应该是没有什么可惊讶的。

讨论

如果模式匹配了一个潜在可能是 null 的值，就需要在匹配过程将 null 处理成非缺省值，这种情况就必须显式地检查 null 值：

```
def printLengthIfString(a: AnyRef): Unit = a match {
  case null => println("Got null!")
```

Null 和 null

因为 Null 不是值类型的子类型，所以 null 也不是从 AnyVal 继承来的任何类型的有效值。这意味着它不可能将 null 值分配给一个变量类型，如 Boolean 或 Int。即使特别指定 null 是 Null 的一个实例，你也不能显式地测试它：

```
scala> val n = null
n: Null = null

scala> n.isInstanceOf[Null]
<console>:8: error: type Null cannot be used in a type
pattern or isInstanceOf test
```

---

① 出自《Java 语言规范》，第 7 版，15.20.2 节：在运行时，如果关系表达式的值不是 null 而且将引用投影到 ReferenceType 而不会发生 ClassCastException 异常，instanceOf 操作的结果就是 true，否则结果是 false。

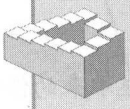
这是因为 Null 在 Java 运行时并不存在。

```
  case str: String =>
      println(s"String of length ${str.length}")
  case _ => println("Not a string")
}
```

另一方面，在 Scala 中利用 Option 工厂是个好的做法，如果参数是 null 就返回 None。

```
scala> def objFromJava: Object = "string"
objFromJava: Object

scala> def stringFromJava: String = null
stringFromJava: String

scala> Option(objFromJava)
res0: Option[Object] = Some(string)

scala> Option(stringFromJava)
res1: Option[String] = None
```

Option 允许你在本来要检查 null 的地方可以检查 None，从而可以将 NullPointerExceptions 转换成编译错误。用 option 可以将 for 表达式里参与可选的值串起计算，而不是在嵌套的 if-else 语句中测试 null。



记住，Scala 中分类模式匹配的是运行时类型而不是编译时类型。结果是它们不会匹配 null，因为 null 不是除 Null 以外任何类型的实例。增加对 null 的显式匹配来操作 null，或者更好的方法是在 option 中将可能返回 null 的任何方法调用的结果打包，这样就可以检查 None 而不是 null。

第 28 章

AnyVal

Scala 的抽象类型成员可以让你定义基本类，而不必立即实现它。比如，你可以定义一个 Recipe 而不必事先决定你需要总数有多精确，比如，这个数量到底用哪种数字类型：

```
trait Recipe {
  type T <: AnyVal
  def sugarAmount: T
  def howMuchSugar() {
    println(s"Add ${sugarAmount} tablespoons of sugar")
  }
}

val approximateCake = new Recipe {
  type T = Int
  val sugarAmount = 5
}

scala> approximateCake.howMuchSugar()
Add 5 tablespoons of sugar

val gourmetCake = new Recipe {
  type T = Double
  val sugarAmount = 5.13124
}

scala> gourmetCake.howMuchSugar()
Add 5.13124 tablespoons of sugar
```

如果要在基本类中初始化抽象类型变量，通常不能用一个特定的值，因为类型本身并不知道。不过，你可以通过设置成_（下划线）将 var 初始化成该类型的缺省值[①]。

① Odersky，《Scala 语言规范》，4.2 节。[Ode14]

让我们看看在 REPL 中执行下面的代码会是什么结果呢？

```
trait NutritionalInfo {
  type T <: AnyVal
  var value: T = _
}
val containsSugar = new NutritionalInfo { type T = Boolean }

println(containsSugar.value)
println(!containsSugar.value)
```

可能的结果

1. 打印出：

```
false
true
```

2. 打印出：

```
false
false
```

3. 第一个语句打印出：

```
null
```

第二个抛出异常。

4. 打印出：

```
null
true
```

解释

你可能想知道 Boolean 值是否通过设置成_而被初始化成缺省值了呢，由于某种原因，不是 false 了。或者你可能怀疑：尽管这个变量是一个 Boolean，可它的初始值却是 null，当你试图对这个值取反时就抛出一个 NullPointerException。

事实上，情况大致如此——但并非完全。正确的答案是 4：

```
scala> println(containsSugar.value)
null

scala> println(!containsSugar.value)
true
```

一个布尔值怎么会是 null 呢？毕竟编译器在初始化时就“知道”变量是个 AnyVal。即便编译器在内部使用 null，可是无论什么原因，为什么这对程序可见了呢？为什么当你试图对这个值取反的时候却没有看到 NullPointerException 呢？

看上去似乎有些令人吃惊，将 AnyVal 变量初始化为 null 的事实并不是编译器的某种技巧。这在语言规格中有明确的说明[①]：

不同类型 T 的缺省值如下：

- 0，如，果 T 是 Int 或是它的一个子范围类型。
- 0L，如果 T 是 Long 型。
- 0.0f，如果 T 是 Float。
- 0.0d，如果 T 是 Double。
- false，如果 T 是 Boolean。
- ()，如果 T 是 Unit。
- null，所有其他的 T 类型。

即便编译器在初始化时就知道变量是 AnyVal 的某个子类型，但它并不知道具体是哪种类型。根据语言规范，null 的确是这种情况最合适的缺省值。

此时如果对这个值取反，编译器当然知道这是一个 Boolean 类型。的确，为了能在它上面调用 unary_!方法，你必须把它当作一个 Boolean 值（Scala 的 boolean 取反运算）。这个方法是在 scala.Boolean 中定义的，并非是从任何 Boolean 的超类型继承来的。

当调用 unary_!时，编译器在底层通过自动拆箱 scala.runtime.BoxesRunTime.unboxToBoolean 把 java.lang.Object（值是 null）当作一个 Boolean 来处理。

---

① Odersky，《Scala 语言规范》，4.2 节。[Ode14]

这个方法通过返回 false 处理一个潜在的 null 值，这是 Scala 里的 Boolean 类型的缺省值。对这个值取反就会打印出“true”。到目前为止，没有什么引人注意的。

那么，第一个 println 语句会怎么做呢？此时，编译器知道这个值也是一个 Boolean 类型。可是为什么你看到的是 null 而不是期望的 Boolean 类型的缺省值 false 呢？

其原因是 println 所期望的参数类型是 Any。当编译器遇到表达式 println(containsSugar.value)时，就要检查 java.lang.Object(containsSugar.value 的类型）类型的一个实例是否能传递给 println。因为 println 只期望一个 Any 就行。这种情况不需要将这个值当成 Boolean 来处理，所以没有应用拆箱，因而潜在的 null 值就被打印出来。

讨论

令人惊讶的是，如果你强制将那个值当成一个 AnyVal 处理也会是这种情况。只有把它当成 Boolean 处理才会应用拆箱：

```
def printAnyVal(a: AnyVal) { println(a) }

scala> printAnyVal(containsSugar.value)
null

def printBoolean(b: Boolean) { println(b) }

scala> printBoolean(containsSugar.value)
false
```

当调用从 java.lang.Object 继承的方法时，如果将值处理成 Any 编译器会拒绝拆箱，这会引起令人惊讶的 NullPointerException 异常：

```
scala> containsSugar.value equals false
java.lang.NullPointerException
  ...
scala> containsSugar.value.hashCode
java.lang.NullPointerException
  ...
scala> containsSugar.value.toString
java.lang.NullPointerException
  ...
```

你可以用 Scala 的 null 安全版本，而不是 equal 和 hashCode：

```
scala> containsSugar.value == false
res11: Boolean = true

scala> containsSugar.value.##
res12: Int = 1237
```

在==的情况，拆箱 containsSugar.value 由 Boolean 类上的==(x: Boolean)方法的出现而被触发，它比 Boolean 变量从 any 继承来的==(arg0: Any)变量更特定[①]。这就迫使编译器将你的值当成 Boolean 处理[②]。

如果你真想拆箱还能用一个类型归属强制编译器拆箱你的值：

```
scala> (containsSugar.value: Boolean) equals false
res13: Boolean = true

scala> (containsSugar.value: Boolean).hashCode
res14: Int = 1237
```

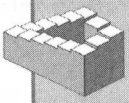
注意，这并不是一个投影，因而没有类型安全的损失。

一个更加严格的解决方案是避免初始化变量直到特定的 AnyVal 子类型已知时。这允许编译器选择合适的缺省值：

```
trait NutritionalInfoNoDefault {
  type T <: AnyVal
  var value: T
}
val containsSugar2 = new NutritionalInfoNoDefault {
  type T = Boolean
  var value: T = _
}

scala> containsSugar2.value equals false
res15: Boolean = true

scala> containsSugar2.value = true
containsSugar2.value: containsSugar2.T = true
```

① Odersky，《Scala 语言规范》，6.26.3 节。[Ode14]

② 关于拆箱的更详细的讨论，请参见第 22 章。



1. 如果这些类型稍后会被固定为 AnyVal 的子类型，要避免用_将抽象类型的变量初始化为它们的缺省值。

2. 如果不能避免，也要小心这个变量可能是 null，无论在哪里，编译器并不需要将它们拆箱成声明的类型。优选 Scala 的 null 安全的方法==和##胜过 equal 和 hashCode，因为它们可以在调用 toString 之前强制拆箱。

第 29 章

隐式变量

Scala 的隐式变量为访问上下文特定的值和行为提供了灵活的机制。通过改变范围内的隐式变量，你能很容易地在应用的不同部分切换上下文。

这段代码建模了一个简单的行李扫描器，它可以在两种模式（或上下文中）进行操作：正常模式和特殊的“测试模式”。在正常模式下，扫描器控制台指示需要扫描哪种类型的物品，且报警按钮是“活动状态的”，一旦激活就会立即触发报警。为了确保操作者能持续保持警觉，扫描器还有一个以随机的间隔被激活的测试模式。在测试模式下，控制台忽视扫描器内实际的物品，并假装发现了危险品。如果操作者点击了报警按钮，报警器就会如期望的那样祝贺他们测试成功。

对应两个上下文，正常模式和测试模式，扫描器的行为由两个隐式变量、一个控制台和一个处理程序定义。当物品通过时，扫描器从正常模式切换到测试模式。

让我们看看在 REPL 中执行下面的代码会是什么结果吧！

```
object Scanner {
  trait Console { def display(item: String) }
  trait AlarmHandler extends (() => Unit)

  def scanItem(item: String)(implicit c: Console) {
    c.display(item)
  }
  def hitAlarmButton()(implicit ah: AlarmHandler) { ah() }
}
object NormalMode {
  implicit val ConsoleRenderer = new Scanner.Console {
    def display(item: String) { println(s"Found a ${item}") }
  }
  implicit val DefaultAlarmHandler = new Scanner.AlarmHandler {
    def apply() { println("ALARM! ALARM!") }
  }
}
```

```
object TestMode {
  implicit val ConsoleRenderer = new Scanner.Console {
    def display(item: String) { println("Found a detonator") }
  }
  implicit val TestAlarmHandler = new Scanner.AlarmHandler {
    def apply() { println("Test successful. Well done!") }
  }
}

import NormalMode._
Scanner scanItem "knife"
Scanner.hitAlarmButton()

import TestMode._
Scanner scanItem "shoe"
Scanner.hitAlarmButton()
```

可能的结果

1. 第 1 个、第 2 个、第 3 个语句打印出：

```
Found a knife
ALARM! ALARM!
Found a detonator
```

第 4 个语句编译失败。

2. 打印出：

```
Found a knife
ALARM! ALARM!
Found a detonator
Test successful. Well done!
```

3. 第 1 个、第 2 个语句打印出：

```
Found a knife
ALARM! ALARM!
```

第 3 个、第 4 个语句编译失败。

4. 打印出：

```
Found a knife
ALARM! ALARM!
Found a shoe
Test successful. Well done!
```

解释

你可能怀疑引入测试模式隐式变量会因为范围内模糊的隐式变量的值引起第 3 和第 4 个语句编译失败。要不然就是 4 个语句都能正常打出结果，或者 4 个语句都会出错？你确定隐式变量的名字会没有影响吗？

实际上，它们确实有影响——正确的答案是 1：

```
scala> Scanner scanItem "knife"
Found a knife

scala> Scanner.hitAlarmButton()
ALARM! ALARM!
...
scala> Scanner scanItem "shoe"
Found a detonator

scala> Scanner.hitAlarmButton()
<console>:17: error: ambiguous implicit values: both value
  DefaultAlarmHandler in object NormalMode of type
    => Scanner.AlarmHandler
  and value TestAlarmHandler in object TestMode of type
    => Scanner.AlarmHandler
  match expected type Scanner.AlarmHandler
              Scanner.hitAlarmButton()
                                     ^
```

怎么会是这样呢？操作者第一次点击报警按钮的时候，编译器能在两个相同类型的隐式变量间进行选择，但是第二次呢？这会与隐式变量值的名字有关吗？如果是，有关系的不是隐式变量的类型而是名字吗？

那么，隐式变量的类型一定决定了它是否可应用在代码的特定位置。为了理解观察到的行为，你需要看看编译器是如何识别和处理多个可应用的选择的。这是一个隐式变量的名字可以起作用的地方。

你可能直觉地期望隐式变量的值应该是某种“特殊的”方式，编译器能像任何其他的 `val` 或 `def` 那样对待它们。因此，当你引入测试模式隐式变量时，`TestMode.ConsoleRenderer` 覆盖了以前引入的 `NormalMode.ConsoleRenderer`。当编译器为第二次 `scanItem` 调用搜索一个隐式变量 `Console` 时，实际在范围内的只有一个可应用的隐式变量值，所以就编译了这个调用。

然而，两个 AlarmHandler 有不同的名字。在引入测试模式的隐式变量后，两个可应用的选择 NormalMode.DefaultAlarmHandler 和 TestMode.TestAlarmHandler 都在范围内。

编译器随后就应用标准的静态重载解析算法[①]来决定使用一个最特定的隐式变量（换句话说，没有“特别的”隐式变量规则[②]）。因此，这两个选择都不比另一个更特别，从而导致了你观察到的编译器错误。

讨论

显然，只要修改测试模式报警处理程序与你打算替换的那个缺省的报警处理程序有相同的名字可以避免这个问题：

```
object TestMode2 {
  implicit val ConsoleRenderer = new Scanner.Console {
    def display(item: String) { println("Found a detonator") }
  }
  // same name as the alarm handler in NormalMode
  implicit val DefaultAlarmHandler = new Scanner.AlarmHandler {
    def apply() { println("Test successful. Well done!") }
  }
}

...

import TestMode2._

scala> Scanner scanItem "shoe"
Found a detonator

scala> Scanner.hitAlarmButton()
Test successful. Well done!
```

如果你不知道要“重载”的隐式变量的名字（例如，由于你正在从一个库导入），可以从“模糊的隐式变量值”错误信息中找到。总是得识别并跟踪要重载的隐式变量的名字，这不是一个特别令人满意的解决方案。

---

① Odersky，《Scala 语言规范》，6.26.3 节。[Ode14]

② Odersky，《Scala 语言规范》，7.2 节。[Ode14]

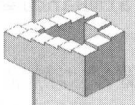
令人高兴的是，Scala 有种方式可以不用知道它们的名字就能重载隐式变量。技巧是确保静态重载解析将重载的隐式变量当成比要被替换的那个更特定的变量。

实现的标准方式是在“重载上下文”所继承的基本类或特质中定义缺省上下文，就像 scala.LowPriorityImplicits 所例示的那样。如下所示：

```
...
class OperatingMode {
  implicit val ConsoleRenderer = new Scanner.Console {
    def display(item: String) { println(s"Found a ${item}") }
  }
  implicit val DefaultAlarmHandler = new Scanner.AlarmHandler {
    def apply() { println("ALARM! ALARM!") }
  }
}

object NormalMode extends OperatingMode


object TestMode extends OperatingMode {
  override implicit val ConsoleRenderer = new Scanner.Console {
    def display(item: String) { println("Found a detonator") }
  }
  implicit val TestAlarmHandler = new Scanner.AlarmHandler {
    def apply() { println("Test successful. Well done!") }
  }
}


import NormalMode._

scala> Scanner scanItem "knife"
Found a knife

scala> Scanner.hitAlarmButton()
ALARM! ALARM!

import TestMode._

scala> Scanner scanItem "shoe"
Found a detonator

scala> Scanner.hitAlarmButton()
Test successful. Well done!
```

在本例代码的这个版本中，第二次调用 hitAlarmButton 时，编译器试图决定最特定的可应用的隐式变量，此时测试模式的操作处理程序更特定一些。用语言规范解释，“TestAlarmHandler 在 TestMode 对象中定义，而 TestMode 继承了 OperatingMode[①]类，OperatingMode 定义了 DefaultAlarmHandler。”

不过，要记住，为使这种方式工作，缺省上下文必须要显式写成被继承的。如果缺省的隐式变量是从你不能控制的代码导入的，例如一个外部库，你就只能寄希望于代码的作者遵从了这种模式。



1. 隐式变量的名字是有关系的！为已存在的隐式变量导入一个相同名字和类型的隐式值，将会从编译器的可应用选项集合中删除现有的隐式变量。导入一个相同类型但不同名字的隐式变量，会导致“模糊的隐式变量值”的编译错误。

2. 当定义一套打算可重载的隐式变量集时，要在可继承的“缺省上下文”类或特质中声明它们。在缺省上下文的子类或子特质中定义重载隐式变量。在这种情况下，更高优先级的隐式变量不必与它们打算替换的隐式变量有相同的名字。

① Odersky，《Scala 语言规范》，6.26.3 节。[Ode14]

第 30 章

显式声明类型

Scala 强大的类型推断功能允许你在许多位置省略掉类型声明，由编译器去推断出类型。而对于非平凡表达式，显式指定类型是个好的做法。

下面这段代码定义了一个从数字字符串转到整数的隐式转换函数的两个版本：第一个没有显式类型声明，第二个有。Println 语句依赖后面的两个隐式变量。让我们看看执行下面的代码会是什么结果吧！

```
class QuietType {
  implicit val stringToInt = (_: String).toInt
  println("4"- 2)
}
class OutspokenType {
  implicit val stringToInt: String => Int = _.toInt
  println("4" - 2)
}

new QuietType()
new OutspokenType()
```

可能的结果

1. 打印出：

```
2
2
```

2. 第 1 个语句编译失败，第 2 个打印出：

```
2
```

3. 两个语句都编译失败。

4. 第 1 个语句打印出：

```
2
```

第 2 个语句抛出运行时异常。

解释

你可能会问为 QuietType.stringToInt 推断出来的类型和为 OutspokenType.stringToInt 显式声明的类型真是一样的吗？确实是一样的：

```
// in QuietType
scala> implicit val stringToInt = (_: String).toInt
stringToInt: String => Int = <function1>

// in OutspokenType
scala> implicit val stringToInt: String => Int = _.toInt
stringToInt: String => Int = <function1>
```

验证了这点之后，你可能又要怀疑是不是隐式变量不可应用到 prinln 语句从而导致编译器错误呢。你确定显式地声明了编译器返回类型（如果不指定编译器也会推断编译器返回类型）不会影响代码的行为吗？

实际上，的确如此——正确的答案是 4：

```
scala> new QuietType()
2

scala> new OutspokenType()
java.lang.StackOverflowError
  at OutspokenType$$anonfun$1.apply(<console>:8)
```

栈溢出错误？怎么会发生这个错？为了理解构造 OutspokenType 的实例时发生了什么，让我们先看看 QuietType 会做什么。尤其是编译器解析了 QuietType.stringToInt 声明的开始部分并正试图理解表达式(_:String). toInt 时，放大这个时刻。此时，stringToInt 的类型还未决定，因为编译器还在试图弄清楚它的过程中。

遗憾的是，编译器很快遇到一个问题：事实上 String 上没有 toInt 方法。根据语言规范[①]，编译器开始在全部范围内[②]的隐式变量中搜索，果真发现了一个

① Odersky，《Scala 语言规范》，6.26 节。[Ode14]

② 关于隐式解析的更详细的讨论，请参见第 29 章。

适合的选项：Predef.augmentString，它将字符串转换成 StringOps 对象，而在 StringOps 上可以调用 Toint。

然后成功地编译隐含的 stringToInt 方法。将它赋值给类型 String => Int，并在处理随后的 println("4" - 2)语句时被成功地发现和应用。

截至目前一切都还好。那么是什么引发了 OutspokenType 表现出如此异常的行为呢？再次聚焦在编译器准备处理 OutspokenType.stringToInt 程序体里的 _.toInt 表达式的时刻。像之前一样，编译器需要调用一个隐式搜索，因为 String 没有 toint 方法。关键的区别就在这里：因为 OutspokenType.stringToInt 的类型是已知的，它在隐式要考虑的列表中。好在它不仅在列表中，而且是可应用的，因为 Int 也恰巧有一个 toInt 方法。

实际上，结果不是像期望的那样转到 StringOps，而是立即调用它自己的隐式变量。我们观察到的结果是无限循环。

讨论

有个一直到现在还没有检查的重要细节是：既然两个隐式变量都在范围内，那么为什么编译器偏偏在 OutspokenType 中选择了 stringToInt 而不是 augmentString 呢？令人惊讶的是，这并没决定应用哪个隐式变量的静态重载解析的结果[①]。事实上，如果你做一个小小的改变，声明 stringToInt 为隐式 def 而不是 val，编译器的确会报出模糊的隐式变量错误：

```
scala> class OutspokenType2 {
         implicit def stringToInt(s: String): Int = s.toInt
         println("4" 2)
       }
<console>:8: error: type mismatch;
 found : s.type (with underlying type String)
 required: ?{def toInt: ?}
Note that implicit conversions are not applicable because
  they are ambiguous:
 both method augmentString in object Predef of type
   (x: String)scala.collection.immutable.StringOps
 and method stringToInt in class OutspokenType2 of type
   (s: String)Int
 are possible conversion functions from s.type
```

① Odersky，《Scala 语言规范》，7.2 节。[Ode14]

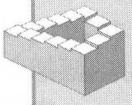
```
    to ?{def toInt: ?}
          implicit def stringToInt(s: String): Int = s.toInt
                                                     ^
```

简而言之：在隐式变量搜索时函数 val 要优先于 def。关键是语言规范并没有证明这个行为，语言规范将静态重载解析默认为决定隐式变量优先级的算法。然而，重载解析同等对待“常规的”def 和函数 val，不会像我们看到的那样表现出隐式变量对函数 val 的偏好。

```
scala> object DefAndFunVal extends App {
         def method(s: String): Int = ???
         val method: String => Int = ???
         println(method("hello"))
       }
<console>:11: error: ambiguous reference to overloaded
  definition,
both value method in object DefAndFunVal of type
  => String => Int
and method method in object DefAndFunVal of type
  (s: String)Int
match argument types (String) and expected result type Any
         println(method("hello"))
                 ^
```

静态重载解析期间，当分辨 def foo:T 和 val foo:() => T 时，Scala 的行为还是未定义的：语言规范说，编译器既不会为了显式方法平等地对待它们，也不会为了隐式方法优选 val。Scala 编译器团队的一个成员 Jason Zaugg 是这样解释的：

在方法和为空方法（如没有参数的方法）之间使用隐式和显式静态重载解析返回函数类型，都是与特定的实施有关的，语言规范没有涵盖这点。



定义隐式转换时，要优选 def 而不是 val，确保实现特定的编译器行为不会隐藏潜在的“模糊的隐式变量”错误；要显式声明隐式 def 的返回类型以改进可读性和安全性。

第 31 章

View

在 Scala 中，往往有多种方式来完成一个特定的任务。下面是一个转换 map 元素的例子，这里用了 for 表达式的方式：

```
def translate(m: Map[String, String]): Map[String, Int] =
  for ((k, v) <- m) yield (k, v.length)
```

这实际上是用 case 表达式来提糖[①]map 调用：

```
def translate(m: Map[String, String]): Map[String, Int] =
  m map { case (k, v) => (k, v.length) }
```

如果只转换 Map 值，模式变量 K 实际上没有用，如例中所示。在这个例子中，mapValues 方法提供了一个更加优雅的选择：

```
def translate(m: Map[String, String]): Map[String, Int] =
  m mapValues (_.length)
```

下面的程序演示了如何使用 map，而且这是一个更简洁的 mapValues 方法。让我们看看它做了什么吧！

```
val ints = Map("15" ->List(1, 2, 3, 4, 5))
val intsIter1 = ints map { case (k, v) => (k, v.toIterator) }
val intsIter2 = ints mapValues (_.toIterator)

println((intsIter1("15").next, intsIter1("15").next))
println((intsIter2("15").next, intsIter2("15").next))
```

可能的结果

1. 打印出：

① 有关提糖一个 for 表达式的更多的例子，请参见第 12 章。

```
(1,2)
(1,2)
```

2. 打印出：

```
(1,1)
(1,1)
```

3. 两个 `println` 语句都抛出异常 `NoSuchElementException:next on empty iterator`。

4. 打印出：

```
(1,2)
(1,1)
```

解释

初看起来似乎两个语句都应该产生相同的输出，比如，结果 map 应该是一致的。因而，第 1 个候选答案看上去最合理。然而，实际上结果却不是 1。正确答案是 4。

```
scala> println((intsIter1(1).next, intsIter1(1).next))
(1,2)

scala> println((intsIter2(1).next, intsIter2(1).next))
(1,1)
```

尤其是第二次调用 `intsIter2(1).next` 结果是 1。能在同一个元素上迭代 `intsIter2(1)` 两次吗？不能，调用 `next` 确实提升了迭代器。那么第二次调用 `intsIter2(1).next` 是如何产生与第一次调用一样的值的呢？只有一个解释：第二次调用 `intsIter2(1)` 返回了一个与第一次不同的迭代器。换句话说，通过某种方式，每次从 `intsIter2` 检索一个值的时候就创建了一个新的迭代器。

`mapValues` 的文档确认了的确是这种情况。它的描述很简单：“通过对每个值执行一次函数来转换 map”，重要的详情隐藏在 return 部分：

`[mapValues returns]`，这是一个将 map 的每个 key 映射到 `f(this(key))` 的 map 视图[①]。其结果是 map 不拷贝任何元素就将原来的 map 打包。

结果是，每次从打包的 map 检索都会重新评估映射函数 `f`，从而创建一个新的迭代器。这与 `map intsIter1` 形成鲜明对比，`map intsIter1` 是 map 转换

① 视图是一种特殊的集合，能将一个或多个转换用 lazily 方式应用于底层“基本”集合。

时只创建一次迭代器，所以每次调用 next 都是在同一个迭代器上执行。这可以用以下代码证明：

```
scala> intsIter1(1) eq intsIter1(1)
res0: Boolean = true

scala> intsIter2(1) eq intsIter2(1)
res1: Boolean = false
```

讨论

从 mapValues 这个名字看不出来它返回的是个视图：

```
def mapValues[C](f: (B) => C): Map[A, C]
```

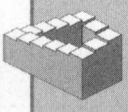
用一个更特定的返回类型或更明显的名字可以更清晰表明，结果实际上是原来 map 上的一个视图，如 mapValuesView。

指出的是，视图可能会存在性能隐患，例如由 mapValues 返回的结果。因为检索每个 map 值都会执行一次函数应用，如果函数太复杂的话，重复地在这样一个 lazy map 上迭代可能会引起相当的负担。

通常你能在视图上调用 force 方法来获得一个严格的集合，但是由 mapViews 返回的类型 collection.immutable.Map 并不提供 force 方法。为了临时权变地解决这个问题，需要在调用 force 之前显式地将 map 转变成一个视图。这会产生期望的结果：

```
val intsIter2 = ints mapValues (_.toIterator)
val strict = intsIter2.view.force

scala> println(strict(1).next, strict(1).next)
(1,2)
```

该谜题里的这段代码出人意料的行为是不可变数据结构的原因之一，而且 Scala 语言的习惯用法也强烈推荐使用纯函数。



尽可能熟悉视图与常规集合之间的不同，小心 mapValues 会返回一个视图，仅从它的名字或特征是看不出来这是个视图的。

第 32 章

toSet

Scala 集合库提供了便利的方法，可以很容易地在不同的集合类型之间进行转换，如 toMap，toList 等。下面这段程序展示这样的一个实例。让我们看看它会做什么吧！

```
val numbers = List("1", "2").toSet() + "3"
println(numbers)
```

可能的结果

1. 打印出：

```
Set(1,2,3)
```

2. 编译失败。

3. 打印出：

```
false3
```

4. 打印出：

```
123
```

解释

候选项看上去最可能的也许是第 1 个：

```
Set(1, 2, 3)
```

如果第 1 行是下面这样，那么选项 1 就的确是正确答案：

```
val numbers = List("1", "2").toSet + "3"
```

然而，原来的程序产生一个完全不同的结果。

```
scala> val numbers = List("1", "2").toSet() + "3"
numbers: String = false3
```

正确的答案是 3。toSet 之后额外的圆括号是两个语句之间的唯一不同之处。这个不同怎么会产生如此不同的结果呢？

让我们仔细查看编译器是如何解析第一个语句的。List[A]上的 toSet 方法明确地将 list 转换成一个 set，但是仔细看 Scala 文档就会揭示出两个重要的事实：

```
def toSet[B >: A]: Set[B]
```

第一，方法本身没有参数。回想一下在 Scala 中，如果一个方法定义时没有圆括号，那么调用时也不能有圆括号：

```
scala> def noParens = "foo"
noParens: String

scala> noParens
res1: String = foo

scala> noParens()
<console>:1: error: not enough arguments for method apply:
  (index: Int)Char in class StringOps.
Unspecified value parameter index.
              noParens()
                       ^
```

第二，产生结果的 set 的元素类型可以不同于原始的 list。更具体地说，set 元素的类型可以是 list 元素类型的超类型。

你能显式地提供期望的元素类型：

```
scala> List("1", "2").toSet[AnyRef]
res1: scala.collection.immutable.Set[AnyRef] = Set(1, 2)
```

副作用的方法

定义参数列表为空的方法在调用时可以有圆括号，也可以没有圆括号：

```
scala> def withParens() = "bar"
withParens: ()String
scala> withParens
```

```
res0: String = bar
scala> withParens()
res1: String = bar
```

可以考虑在方法调用中包括空的圆括号。

不用说，如果没有提供类型参数，编译器就得推断出其类型。通常，结果集合的元素类型与原集合的元素类型相同，这是很明显的：

```
scala> List("1", "2").toSet
res2: scala.collection.immutable.Set[String] = Set(1, 2)
```

然而，这种情况下，附加的圆括号就会让编译器寻找不同的原因。因为 toSet 不能用圆括号调用，编译器只能将圆括号解释成试图在 toSet 的调用结果上调用 apply。巧合的是，特质 trait Set[A]恰好有一个 apply 方法，这等同于方法 contains：

```
def apply(elem: A): Boolean
```

它测试某个元素是否包含在这个集合中。

因而，编译器就会将带圆括号的 toSet 调用处理成：

```
List("1", "2").toSet.apply() // apply() == contains()
```

但是这个例子中要被测试的元素是什么呢？apply 方法明确期望一个参数（它的参数 elem 并没有一个缺省值），看起来好像提供了 none。这里一个叫作参数适配器的编译器特性起作用了——编译器适配参数列表，插入 Unit 值，()，匹配由方法声明指定的单个参数。你能用-Ywarn-adapted-args 选项[①]通过编译相同的代码行来验证这点：

```
scala> List("1", "2").toSet.apply()
<console>:8: warning: Adapting argument list by inserting ():
  this is unlikely to be what you want.
        signature: GenSetLike.apply(elem: A): Boolean
  given arguments: <none>
after adaptation: GenSetLike((): Unit)
              List("1", "2").toSet.apply()
```

作为参数适配器的结果，最后的表达式实际上是：

```
List("1", "2").toSet.apply(())
```

① 从 Scala 2.11 开始，这个行为不再支持，编译器警告是缺省发出的。有关-Ywarn-adapted-args 更详细的讨论和相关的编译器选项，请参见第 26 章。

此时，编译器必须得决定由 toSet 返回的 set 是否含有 Unit 值。因为 Unit 值明显不是一个 String，所以这怎么会编译呢？回想一下，编译器能推断出结果集的元素类型是 String 的任何超类型[①]，如 Scala 类型层次结构[②]所示，Unit 和 String 公共的超类型是 Any。结果表达式就是：

```
List("1", "2").toSet[Any].apply(())
```

令人惊讶的是，这个表达式评估为 false，原因是 Unit 值并不在 List 里。因此，实际被评估的表达式是：

```
false + "3"
```

因为 Scala 像 Java 一样，允许 String 能与任何其他类型连接，所以这个表达式会编译且结果是字符串"false3"[③]。

参数适配器

自动插入 Unit 值，( )，这实际上是 Scale 的自动生成元组的一个非本意的结果。自动生成元组允许编译器在只期望一个参数的地方将多个方法参数打包成一个元组，例如，假设有以下方法：

```
def tell(o: Any): Unit
```

用 3 个参数的调用：

```
tell(a, b, c)
```

编译器将 3 个参数打包进一个元组：

```
tell((a, b, c))
```

如果没有提供参数：

```
tell()
```

编译器也会执行一个相似的转换：

```
tell(())
```

这里编译器试图插入一个“空元组”，这恰好与 Unit 值匹配[④]。

---

① 有关的讨论，请参见第 25 章。

② 参见图 27.1。

③ 有关的讨论，请参见第 36 章。

④ 有关的讨论，请参见第 26 章。

讨论

你已经看到调用 toSet 之后省略圆括号产生了期望的结果。注意，尽管如此，如果你希望跟踪代码看看表达式是如何一步步被解析的、会推断出什么类型，你会发现在 REPL 中很难实现：

```
scala> val list = List("1", "2")
list: List[String] = List(1, 2)

scala> val set = list.toSet
set: scala.collection.immutable.Set[String] = Set(1, 2)

scala> val result = set()
<console>:9: error: not enough arguments for method apply:
  (elem:String)Boolean in trait GenSetLike.
Unspecified value parameter elem.
       val result = set()
                    ^
```

这里最后一个语句编译失败，是因为将 Set[String]转成 Set[Any]的类型加宽并没有发生。也就是说，set 的元素类型已经被推断为 String，所以它不再可能去检查那个明确不是一个 String 的 Unit 值是否包含在集合里。

另一方面，在例子程序的第一行中，由于类型推断总是检查整个表达式，所以它能推断出 Any 为结果集的元素类型。

如果要保留原来的集合类型，还可以用更通用的转换方法：定义在 List[A]上（或任何 Traversable[A]或类似的）to 方法，其特征如下：

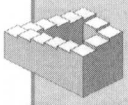
```
def to[Col[_]]: Col[A]
```

调用这个方法而不是 toSet 会导致编译器错误：

```
scala> List("1", "2").to[Set]() + "3"
<console>:7: error: not enough arguments for method to:
  (implicit cbf: scala.collection.generic.CanBuildFrom[
  Nothing,String,Set[String]])Set[String].
Unspecified value parameter cbf.
              List("1", "2").to[Set]() + "3"
                                       ^
```

与预期的一样，下面这个调用成功了（例如，没有圆括号）：

```
scala> List("1", "2").to[Set] + "3"
res0: scala.collection.immutable.Set[String] = Set(1, 2, 3)
```



在方法调用中用空的圆括号只是因为副作用的方法才这样做。

由于集合上的方法允许返回的集合的元素类型比原来的元素类型宽，所以要小心由集合上的方法引起的非本意的类型加宽。

第 33 章

缺省值

在许多语言里，给 map 中的数据项赋以缺省值都需要一段繁琐冗长的代码：

```
import collection.mutable
val accBalances = mutable.Map[String, Int]()

// opening credit is linked to the account holder's name
def getBalance(accHolder: String): Int = {
  if (!(accBalances isDefinedAt accHolder)) {
    accBalances += (accHolder ->accHolder.length)
  }
  accBalances(accHolder)
}

scala> println(getBalance("Alice"))
5
```

好在 Scala 提供了一个缺省函数就能轻松实现（也要避免可变的 map）

```
import collection.immutable
val accBalances = immutable.Map[String, Int]() withDefault {
  newCustomer => newCustomer.length }

scala> println(accBalances("Bob"))
3
```

这是提供基于 map key 缺省值的一种很好的清晰方式。通常缺省值并不依赖 key，所以，在这个例子中，用 Map 的 withDefaultValue 方法更合适：

```
import collection.immutable
val accBalances =
  immutable.Map[String, Int]() withDefaultValue 10

scala> println(accBalances("Bob"))
10
```

在下面的例子中，每个账号拥有者以“感谢您的参与”开始一个余额为 100 美元的账号。账号以两种不同的方式表示，简单的余额信息在 accBalances 里，余额历史信息在 accBalancesWithHist 里。两个新客户来兑现他们意料之外获得的礼物。让我们看看执行下面的代码会是什么结果吧！

```
import collection.mutable
import collection.mutable.Buffer

val accBalances: mutable.Map[String, Int] =
  mutable.Map() withDefaultValue 100

def transaction(accHolder: String, amount: Int,
    accounts: mutable.Map[String, Int]) {
  accounts += accHolder ->(accounts(accHolder) + amount)
}

val accBalancesWithHist: mutable.Map[String, Buffer[Int]] =
  mutable.Map() withDefaultValue Buffer(100)

def transactionWithHist(accHolder: String, amount: Int,
    accounts: mutable.Map[String, Buffer[Int]]) {
  val newAmount = accounts(accHolder).head + amount
  accounts += accHolder ->
    (newAmount +=: accounts(accHolder))
}

transaction("Alice", 100,accBalances)
println(accBalances("Alice"))
println(accBalances("Bob"))
transactionWithHist("Dave", 100,accBalancesWithHist)
println(accBalancesWithHist("Carol").head)
println(accBalancesWithHist("Dave").head)
```

可能的结果

1. 打印出：

```
-100
0
0
-100
```

2. 打印出：

```
0
100
0
100
```

3．打印出：

```
0
100
100
0
```

4．打印出：

```
0
100
0
0
```

解释

你可能想知道在检索任何值之前先更新 map 是否会以某种方式影响缺省值呢，或者是否因数据项的访问顺序而不同呢。那么给账户增加历史确定没有影响吗？

的确有影响。正确的答案是 4 不同：

```
scala> println(accBalances("Alice"))
0

scala> println(accBalances("Bob"))
100

scala> println(accBalancesWithHist("Carol").head)
0

scala> println(accBalancesWithHist("Dave").head)
0
```

所以 Alice 和 Bob 的账户余额都如所期望的，但 Carol 和 Dave 的账户历史却不像期望的那样。为了理解发生了什么，让我们比较 Carol 和 Dave 实际的账户对象：

```
scala> println(accBalancesWithHist("Carol")
```

```
        eq accBalancesWithHist("Dave"))
true
```

换句话说，一旦增加了账户历史，Carol 和 Dave（以及所有其他账户拥有者）就在共享同一个账户！withDefault 的行为可能会让你认为 withDefaultValue 是作为一个“工厂”为每个 map 数据项提供新的缺省值的实例。事实并非如此：所有数据项的缺省值都是同一个值。

讨论

因为在所有的 map 数据项之间共享相同的值会有无法预料的结果，所以最安全最可预测的提供缺省值的方法是用 withDefault，当然是为可变的（mutable）数据项提供缺省值。这为每个 map 数据项创建了一个缺省值的新的实例。

```
val accBalancesWithHist2: mutable.Map[String, Buffer[Int]] =
  mutable.Map() withDefault { _ => Buffer(100) }

transactionWithHist("Dave", 100, accBalancesWithHist2)

scala> println(accBalancesWithHist2("Carol").head)
100

scala> println(accBalancesWithHist2("Dave").head)
0
```

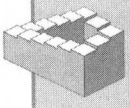
如果你要避免总是要对 withDefault 缺省函数省去下划线（_）参数这样的样板代码，或许用一个方法名显式提到值的方法能改进可读性，那么你就可以定义自己的 withDefaultValue 方法。

```
port collection.mutable
import collection.mutable.Buffer
implicit class MapDefaults[A, B](
    val map: mutable.Map[A, B]) extends AnyVal {
  def withNewDefaultValue(d: => B): mutable.Map[A, B] =
    map withDefault { _ => d }
}

...

val accBalancesWithHist: mutable.Map[String, Buffer[Int]] =
  mutable.Map() withNewDefaultValue Buffer(100)
```

```
transactionWithHist("Dave", 100, accBalancesWithHist)

scala> println(accBalancesWithHist("Carol").head)
100

scala> println(accBalancesWithHist("Dave").head)
0

scala> println(accBalancesWithHist("Carol")
         eq accBalancesWithHist("Dave"))
false
```

这里，声明的值以 by-name 参数[①②]传递给 withNewDefaultValue，使它在每次缺省值函数被调用的时候都要重新评估。这会使每个 map 数据项都被赋值给一个“新”的缺省值的实例。

如果缺省值是不可变的，在所有 map 数据项之间共享相同的实例就不是问题。事实上，如果构建值的成本比较高，这种方式就比为每个 map 数据项都创建一个新实例成本要低。



只对不可变的缺省值才用 Map.withDefaultValue，否则用 Map.withDefault。

要注意 withDefaultValue 会导致缺省值的同一个实例在所有的 map 数据项之间共享。

① Odersky，《Scala 语言规范》，4.6.1 节。[Ode14]

② 关于 by-name 参数的更详细的讨论，请参见第 23 章。

第 34 章

关于 Main

Scala 的一个便利特性是能够将初始化语句直接放在类、特质或对象体中，而不一定非得定义一个显式的主构造器：

```
class HelloWorld {
  val msg = "Hello World!"
  println(msg)
}

scala> new HelloWorld
Hello World!
```

Scala 利用这个特性可以简单地定义一个能从命令行[①]运行的程序，而不是一定要定义一个有 main 方法的程序，一个对象能简单地从 App 特质继承。对象体能自动变成 main 方法的内容并在程序运行时执行：

```
object HelloWorld extends App {
  println("Hello World!")
}

scala> HelloWorld main Array()
Hello World!
```

下面的例子使用这个特性来重构一个简单的机场模拟程序。在模拟程序的两个版本中，两名旅客带着很重的行李走近办理登机手续处，代理程序根据这名乘客的飞机是否满员做出响应。

模拟程序的第一个版本定义了一个显式的 main 方法，第二个版本通过继承 App 避免使用 main。执行下面的代码会是什么结果呢？

```
class AirportDay {
```

① Odersky，《Scala 语言规范》，9.5 节。[Ode14]

```
  def tryCheckBag(weight: Int): String =
    "It's not a full flight. Your bag is OK."
}
class StartOfVacation extends AirportDay {
  override def tryCheckBag(weight: Int): String =
    if (weight > 25)
      "Your bag is too heavy. Please repack it."
    else
      "Your bag is OK."
}
def goToCheckIn(bagWeight: Int)(implicit ad: AirportDay) {
  println(s"The agent says: ${ad tryCheckBag bagWeight}")
}

object AirportSim {
  def main(args: Array[String]): Unit = {
    implicit val quietTuesday = new AirportDay
    goToCheckIn(26)
    implicit val busyMonday = new StartOfVacation
    goToCheckIn(26)
  }
}
object AirportSim2 extends App {
  implicit val quietTuesday = new AirportDay
  goToCheckIn(26)
  implicit val busyMonday = new StartOfVacation
  goToCheckIn(26)
}

AirportSim main Array()
AirportSim2 main Array()
```

可能的结果

1. 第一个机场模拟程序运行打印出：

```
The agent says: It's not a full flight. Your bag is OK.
The agent says: Your bag is too heavy. Please repack it.
```

第二个抛出异常。

2. 打印出：

```
The agent says: It's not a full flight. Your bag is OK.
The agent says: Your bag is too heavy. Please repack it.
```

```
The agent says: It's not a full flight. Your bag is OK.
The agent says: Your bag is too heavy. Please repack it.
```

3. 打印出：

```
The agent says: It's not a full flight. Your bag is OK.
The agent says: Your bag is too heavy. Please repack it.
The agent says: Your bag is too heavy. Please repack it.
The agent says: Your bag is too heavy. Please repack it.
```

4. 打印出：

```
The agent says: Your bag is too heavy. Please repack it.
The agent says: Your bag is too heavy. Please repack it.
The agent says: Your bag is too heavy. Please repack it.
The agent says: Your bag is too heavy. Please repack it.
```

解释

你可能会问：为什么这两种情况不是都简单地得到一个“模糊的隐式值”错误呢。或者可能是因为 busyMonday 比 quietTuesday 的优先级高些，所以代理程序每次都抱怨行李。你确定从显式的 main 方法改成从 App 继承的方式会得到同样的结果吗？

从 REPL 执行，如你看到的，正确答案是 1：

```
scala> AirportSim main Array()
The agent says: It's not a full flight. Your bag is OK.
The agent says: Your bag is too heavy. Please repack it.

scala> AirportSim2 main Array()
java.lang.NullPointerException
  at .goToCheckIn(<console>:9)
  ...
  at AirportSim2$.main(<console>:10)
  ...
```

抛出 NullPointerException 异常？这是从哪里来的？为了理解这里发生了什么，让我们看看第一个机场模拟程序 AirportSim 里在进行什么。

在 AirportSim.main 里两次调用 goToCheckIn(26) 都必须要一个隐式的 AirportDay。当第一次调用 goToCheckIn 时，只有一个候选 quietTuesday 在范围内，因为它是此时代码里声明的唯一隐式参数。因而，服务台代理很仁慈地就让这名旅客通过了。

当再次调用 goToCheckIn 时，就声明了第二个适合的隐式变量 busyMonday，而且也在范围内。现在，编译器需要通过应用静态重载解析[①]规则决定哪个隐式变量是最特定的[②]。

根据这些规则，StartOfVacation 的确是更特定的隐式变量，因为它从 AirportDay 继承，所以没有“模糊的隐式值”错误和不幸的乘客被告知要重新打包[③]。

然后让我们再看 AirportSim2 的情况有什么不同吗？即便语言规则将 App 描述为“一个其程序体是 main 方法[④]的特别类”，但它的程序体实际上并不是一个方法（而是一个构造器）。所以这里声明的隐式 val 不是一个本地变量，而是对象成员[⑤]。

这意味着两个隐式变量都在对象体的范围内。具体地说，busyMonday 在首次调用 AirportSim2 里的 goToCheckIn 时也是可应用的。此时，busyMonday 是范围内最特定的隐式变量，它被编译器选中。

那么怎么会出现 NullPointerException 异常呢？它的出现是构造器里的语句顺序的结果。隐式 val 声明的位置并没有指示何时 val 会进入范围，但是它却决定这个值实际上何时被初始化。根据语言规则，在 AirportSim2 程序体中的初始化语句确实是按顺序执行的[⑥]。

这意味着范围内最特定的隐式变量 busyMonday 在被传递给第一次调用的 AirportSim2 中的 goToCheckIn(26) 时还没有被初始化。因而是该类型的缺省值 null。当 goToCheckIn 试图调用它时，就看到 NullPointerException 异常的结果。

讨论

构造器里声明的值对整个类或对象都是可见的，并且在它们被初始化之前也能被引用，这个事实可以应用到常规的 val 和 var，还可应用到类和对象。

---

① Odersky，《Scala 语言规范》，7.2 节。[Ode14]
② Odersky，《Scala 语言规范》，6.26.3 节。[Ode14]
③ 相关的讨论，请参见第 29 章。
④ Odersky，《Scala 语言规范》，9.5 节。[Ode14]
⑤ Odersky，《Scala 语言规范》，5.1.3 节和 5.4 节。[Ode14]
⑥ Odersky，《Scala 语言规范》，5.1 节。[Ode14]

```
class HelloWorld {
  println(message)
  val message = "Hello World!"
}

scala> new HelloWorld
null
```

幸运的是，编译器能发现这些引用并发出合适的警告：

```
scala> class HelloWorld {
         println(message)
         val message = "Hello World!"
       }
<console>:8: warning: Reference to uninitialized value
  message
         println(message)
                 ^
```

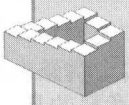
同样，当表达式从一个类的辅助构造器移到主构造器时，出乎意料的是，行为发生了相同的改变。即便辅助构造器也是一个构造器，其行为就像 AirportSim 的 main 方法。辅助构造器中的值声明被当作本地变量，并把构造器表达式当做函数一样评估[①]：

```
class AirportSimAuxiliaryCons {
  def this(weight: Int) {
    this()

    implicit val quietTuesday = new AirportDay
    goToCheckIn(weight)

    implicit val busyMonday = new StartOfVacation
    goToCheckIn(weight)
  }
}

scala> new AirportSimAuxiliaryCons(26)
The agent says: It's not a full flight. Your bag is OK.
The agent says: Your bag is too heavy. Please repack it.
```

当在主构造器中声明的时候，相同的隐式参数值现在是类成员，正如 AirportSim2 的例子里一样。结果第一次调用 goToCheckIn 时它们都在范围内，所以就能更早看到同样的 NullPointerExceptionde 的结果：

① Odersky，《Scala 语言规范》，5.3.1 节。[Ode14]

```
class AirportSimPrimaryCons(weight: Int) {

  implicit val quietTuesday = new AirportDay
  goToCheckIn(weight)

  implicit val busyMonday = new StartOfVacation
  goToCheckIn(weight)
}

scala> new AirportSimPrimaryCons(26)
java.lang.NullPointerException
  at .goToCheckIn(<console>:9)
  ...
```



方法里的本地变量是直到被声明才会在范围内。与之不同是，在类或对象体的顶级声明的变量是成员，它们会在类或对象的整个范围内可见，但是直到运行到声明它们的代码行才会被初始化。

要注意编译器可能会发生“引用未初始化的变量”的警告。所以一定要验证这个引用的确是你打算引用的，还要验证你的代码会正确地处理（未初始化的）缺省值。

第 35 章

列表

允许 Scala 满足不同代码样式需要的一个便利特征是你往往能将大括号换成圆括号：

```
scala> (1 to 3).foreach(r =>
         print("%.5f ".format(math.Pi * r * r)))
3.14159 12.56637 28.27433

scala> (1 to 3) foreach { r =>
         print("%.5f ".format(math.Pi * r * r)) }
3.14159 12.56637 28.27433
```

Scala 也提供类型别名，这允许你将更多的便利名给非平凡类型，并将域特定的名字给通用类型：

```
// letters > terms and the pages on which they appear
type BookIndex = Map[Char, Map[String, Seq[Int]]]

type Fahrenheit = Int
type Celsius = Int
// compare with 'def toFahrenheit(celsius: Int): Int'
def toFahrenheit(c: Celsius): Fahrenheit
```

下面的程序使用了这两个特性。让我们看看它做了什么吧！

```
type Dollar = Int
final val Dollar: Dollar = 1
val x: List[Dollar] = List(1, 2, 3)

println(x map { x: Int => Dollar })
println(x.map(x: Int => Dollar))
```

可能的结果

1. 打印出：

```
List(1, 2, 3)
List(1, 2, 3)
```

2. 打印出：

```
List(1, 1, 1)
List(1, 1, 1)
```

3. 第一个 `println` 语句打印出：

```
List(1, 1, 1)
```

第二个抛出一个异常。

4. 第一个 `println` 语句打印出：

```
List(1, 2, 3)
```

第二个编译失败。

解释

你可能疑惑是否两个语句都会编译呢。如果它们都能编译，你确定在这里无论用大括号还是圆括号都没有区别吗？实际上，的确有区别——正确答案是 3：

```
scala> println(x map { x: Int => Dollar })
List(1, 1, 1)

scala> println(x.map(x: Int => Dollar))
java.lang.IndexOutOfBoundsException: 3
...
```

要点是，匿名函数在块表达式里传递[①]相对于直接传递[②]来说是解析得不同的。为了理解怎么会有观察到的行为，让我们看看函数表达式的两种不同的解释方式：

```
x: Int => Dollar
```

---

① Odersky，《Scala 语言规范》，6.23 节。[Ode14]

② 关于在块表达式中传递匿名函数，在第 1 章中也有所讨论。

一种方式是把它当作一个函数，取出一个 Int 返回值 Dollar，比如常数值 1。换句话说，你可能得出结论 x 是一个参数：

```
(x: Int) => Dollar
```

尽管对于匿名函数来说将参数列表放在圆括号里总是对的，但某些情况也能省略。具体地说，如果一个函数只有一个参数且没有给出类型，那么圆括号就不是必需的，如 x => Dollar。如果指定了参数的类型，就只有在匿名函数出现为块的时候才能省略圆括号。这恰恰就是第一个 println 语句所发生的：

```
println(x map { x: Int => Dollar })
```

这个语句等同于下面这个更简短的形式：

```
println(x map { freshName => 1 })
```

一点也不奇怪，结果是一个 1 的 list：

```
scala> println(x map { x: Int => Dollar })
List(1, 1, 1)
```

注意，这个表达式中两次出现的 x 有不同的含义：第一个 x 指列表，而第二个 x 表示一个传递给 map 函数的参数。

在第二个 println 语句中，匿名函数并不是以块表达式传递的：

```
println(x.map(x: Int => Dollar))
```

假设省略掉围绕参数 x: Int 的圆括号，且表达式 x: Int => Dollar 不在块内，你可能要问它该怎么编译呢。答案是：通过将类型声明当作一个类型归属于整个表达式，就能用不同的方式读到这个函数表达式。换句话说，表达式 x: Int => Dollar 被解析为 x: (Int => Dollar)，x 是一个 Int => Dollar 的函数类型，Dollar 是 Int 的一个类型别名。

这意味着第二个 println 语句中出现的两个 x 都指向同一个值，x 列表。这是唯一可能的原因，因为 Scala 的 List 也是一个 Function1，它的 maps 列表索引到列表中的元素。因此，List x（这是一个 List[Dollar]）是从 Int 到 Dollar 的一个函数，是 map 所期望的类型。这个语句因而会编译成功。

结果是，在运行时，第二个 println 语句使用 list 中的每个值作为一个索引返回它自己。更具体地说，它试图将 List(1, 2, 3) 映射到 List(x(1), x(2), x(3))。因为 list x 仅有 3 个元素，这会一直持续到传递了最后的索引值 3，从而结果就是观察到的 IndexOutOfBoundException 异常。

讨论

为了使直接传递匿名函数与传递块表达式有相同的结果，你需要将参数放在圆括号内：

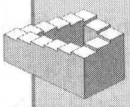
```
scala> println(x.map((x: Int) => Dollar))
List(1, 1, 1)
```

如果没有指定参数类型，以下两个 println 语句会表现出相同的行为：

```
scala> println(x map { x => Dollar })
List(1, 1, 1)

scala> println(x.map(x => Dollar))
List(1, 1, 1)
```

你可以传递一个程序体直接含有一个表达式的匿名函数或者作为一个块传递，下面两种情况都需要一个块表达式：

```
// multiple statements in function body
Seq(1, 2, 3) map { e =>
  val i = e + 1
  i * 2
}

// using case clauses
Seq(1, 2, 3) map {
  case e if e % 2 == 0 => e + 2
  case e => e + 1
}
```



注意：在大括号中传递一个匿名函数实际上是创建了一个块。匿名函数在块中传递会解析得与直接传递不同。

如果你正在用圆括号传递一个参数的匿名函数，如果这个匿名函数含有一个类型声明，就一定要将参数放在圆括号里。

第 36 章

计算集合的大小

Scala 集合的目标之一是进行公用的操作，例如连接两个集合、简洁和容易阅读。为了实现这个目的，集合类型支持大量的操作符，如++和+: 操作符；它们允许你写出“自然的”表达式并避免如 concat 或 prepend 这样的方法名。

以下代码用其中一个这样的操作来判断在一名年轻的 Scala 狂人的午餐盒里有多少种食品。为了确保有一个适度健康的饮食，我们也偷偷地在午餐盒里放了一个苹果。

我们的 Scala 狂人有个特别的嗜好：每样东西只吃一个，所以代码先从确定午餐盒是 set 开始。意识到这点，我们更新了代码：午餐盒里允许选择任意品种，使用原始的确保午餐盒是集合的函数，任何集合都可以更新信息，计算当前的“苹果加强”的午餐盒有多少种食物。执行下面的代码会是什么结果呢？

```
import collection.mutable

def howManyItems(lunchbox: mutable.Set[String],
  itemToAdd: String): Int = (lunchbox + itemToAdd).size

def howManyItemsRefac(lunchbox: mutable.Iterable[String],
  itemToAdd: String): Int = (lunchbox + itemToAdd).size

val lunchbox =
  mutable.Set("chocolate bar", "orange juice", "sandwich")
  println(howManyItems(lunchbox, "apple"))
  println(howManyItemsRefac(lunchbox, "apple"))
  println(lunchbox.size)
```

可能的结果

1. 打印出：

```
4
5
5
```

2. 打印出：

```
4
4
3
```

3. 打印出：

```
4
4
4
```

4. 打印出：

```
4
47
3
```

解释

你可能会问：是否会把午餐盒当成一个 Iterable 来处理呢？是否会像 howManyItemsRefac 的情况一样，允许以某种方式增加第二个苹果呢？或者你可能怀疑：增加一个苹果就创建了一个新的午餐盒，而不是修改传递给计数函数的午餐盒，其结果是会打印出 4，4，3。

如所发生的那样，这个想法比较接近了，但还不完全是这样。正确的答案是 4：

```
scala> println(howManyItems(lunchbox, "apple"))
4

scala> println(howManyItemsRefac(lunchbox, "apple"))
47

scala> println(lunchbox.size)
3
```

这里发生了什么呢？首先：你怀疑传递给计数函数的午餐盒实际上并没有被修改，这是对的。可变的和不可变的 Set 上的+方法都创建了一个新的 set，由原来的 set 中的元素和传递的参数一起组成。实际上修改 set 的可变集合上的

方法是+=。所以 howManyItems 就返回加了苹果的新午餐盒的大小，但实际上新的午餐盒并没有加进苹果。这就解释了为什么第一个 println 语句输出是 4，最后一个 println 打印的却是 3。

howManyItemsRefac 发现的 47 个午餐盒条目又是从哪里来的呢？实际上+并不是 Iterable 支持的方法，而是 Scala 的字符串连接操作符。itemToAdd 确实是个字符串。

当然，lunchbox 不是一个字符串，所以 Scala 编译器开始寻求一个可应用的隐式转换[①]的帮助。它找到一个形如 Predef.any2stringadd[②]的隐式转换。应用这个缺省隐式转换能将任何对象转换成字符串以便它能与其他字符串连接。

本质上，howManyItemsRefac 的函数体是：

```
Predef.any2stringadd(lunchbox).+(itemToAdd).size
```

这意味着增加一个苹果后 howManyItemsRefac 并没有返回午餐盒里的条目数，而是 lunchbox.toString + "apple"的字符串长度：

```
def howManyItemsDebug(lunchbox: mutable.Iterable[String],
    itemToAdd: String): Int = {
  val concatenatedStrings = lunchbox.toString + itemToAdd
  println(s"DEBUG: ${concatenatedStrings}")
  concatenatedStrings.size
}

scala> println(howManyItemsDebug(lunchbox, "apple"))
DEBUG: Set(orange juice, sandwich, chocolate bar)apple
47
```

讨论

any2stringadd 存在的主要原因是为了与 Java 一致。回想一下 Java 也支持将任何对象与字符串连接。这是快速调试语句的实用方法：

```
case class Pet(name: String, species: String)
println(Pet("garfield", "cat") + " is my pet")
```

① Odersky，《Scala 语言规范》，6.26 节。[Ode14]

② 关于隐式解析的相关讨论，参见第 30 章。

由于 any2stringadd 出现在代码中意想不到的位置很容易让人费解，所以不断有人要求把它删掉。将来很可能会删掉，尽管短期还不是太必要。好在通过用 import selector[①]将有问题的隐式“重命名”为匹配符_，就能禁用这个函数或禁用你的程序中确实存在的任何其他缺省隐式。实际上是让它们不可访问。注意，必须把这些 import 语句放在源代码的第一行。

在 REPL 中禁用隐式，可以用:paste raw 命令，它能将随后的输入当作 Scala 文件，而不是一个脚本：

```
scala> :paste raw
// Entering paste mode (ctrlD to finish)
import Predef.{any2stringadd => _, _}
object SizeItUp {
  import collection.mutable
  def howManyItemsRefac(lunchbox: mutable.Iterable[String],
    itemToAdd: String): Int = (lunchbox + itemToAdd).size
}
// Exiting paste mode, now interpreting.
<pastie>:5: error: value + is not a member of
  scala.collection.mutable.Iterable[String]
    itemToAdd: String): Int = (lunchbox + itemToAdd).size
                                          ^
```

这里，import Predef.{any2stringadd => _, _}中的第二个匹配符（_）是确保 Predef 的所有其他成员也被导入。

如果不用这么优雅，还有一个更简单的方式可以禁用隐式转换，就是故意引起一个“模糊的隐式值”冲突。用这种方式，你需要引入两个新的隐式转换。如果只定义了一个隐式转换，它就会比你希望禁用的 Predef 中的转换还要特定且会被选中，而不是产生希望的冲突：

```
object NoAny2StringAdd {
  implicit val disableAny2stringadd1 = (_: Any) => ""
  implicit val disableAny2stringadd2 = (_: Any) => ""
}
import collection.mutable
import NoAny2StringAdd._

scala> def howManyItemsRefac(
           lunchbox: mutable.Iterable[String],
           itemToAdd: String): Int =
```

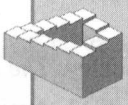
① Odersky，《Scala 语言规范》，4.7 节。[Ode14]

```
        (lunchbox + itemToAdd).size
<console>:13: error: type mismatch;
 found   :lunchbox.type (with underlying type
   scala.collection.mutable.Iterable[String])
 required: ?{def +(x$1: ? >: String): ?}
Note that implicit conversions are not applicable
  because they are ambiguous:
 both value disableAny2stringadd1 in object NoAny2StringAdd
  of type => Any => String
and value disableAny2stringadd2 in object NoAny2StringAdd
  of type => Any => String
are possible conversion functions from lunchbox.type to
  ?{def +(x$1: ? >: String): ?}
        (lunchbox + itemToAdd).size
         ^
```

一种阻止不希望的字符串连接的更加直观的方式是显式声明集合操作的期望类型：

```
scala> def howManyItemsRefac(
         lunchbox: mutable.Iterable[String],
         itemToAdd: String): Int = {
       val healthierLunchbox: mutable.Iterable[String] =
         lunchbox + itemToAdd
       healthierLunchbox.size
     }
<console>:10: error: type mismatch;
 found   :String
 required:scala.collection.mutable.Iterable[String]
         lunchbox + itemToAdd
                  ^
```

在这种方式里，使用一个中间 val 不会引起任何的性能损失：一般说来，编译器能够优化掉这个中间值。



注意，如果参数是个字符串，Scala 编译器总能将+操作符当作字符串连接符。如果表达式不希望返回一个 String，那就显式声明期望的结果类型。你能禁用程序中的 Predef.any2stringadd 来阻止将任何对象隐式转换成一个字符串。

参考文献

[Dou] "(Double.NaN min 0.0) yields 0.0, should be NaN." *Scala Programming Language / SI-5104*. Available on the web at https://issues.scala-lang.org/browse/SI-5104 (accessed November 17, 2014).

[EPF] EPFL. *Scala Standard Library, Scaladoc Documentation*. Available on the web at http://www.scala-lang.org/api/current/ (accessed November 17, 2014).

[Fow99] Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, first edition, 1999.

[Gle11] Gleichmann, Mario. "Functional Scala: Turning Methods into Functions." January 2011. Available on the web at http://gleichmann.wordpress.com/2011/01/09/functional-scalaturning-methods-into-functions/ (accessed November 17, 2014).

[Gri10] Griffith, Dave. "Purpose of 'return' statement in Scala?" September 2010. Available on the web at http://stackoverflow.com/a/3771243/391960 (accessed November 17, 2014).

[Har] Harrah, Mark. "Value Classes and Universal Traits." Available on the web at http://docs.scala-lang.org/overviews/core/valueclasses. html (accessed November 17, 2014).

[Lin13] Lindholm, Tim, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE7 Edition*. February 2013. Available on the web at http://docs.oracle.com/javase/specs/jvms/se7/html/ (accessed November 17, 2014).

[Mat12] Mathias. "The Magnet Pattern." December 2012. Available on the web at http://spray.io/blog/2012-12-13-the-magnet-pattern/(accessed November 17, 2014).

[Odea] Odersky, Martin and Lex Spoon. "Collections: Views." Available on the web at http://docs.scalalang. org/overviews/collections/views.html (accessed November 17, 2014).

[Odeb] Odersky, Martin, Lex Spoon, and Bill Venners. "Programming in Scala, Glossary." Available on the web at http://docs.scalalang. org/glossary/#uniform_access_principle (accessed November 17, 2014).

[Ode10] Odersky, Martin, Lex Spoon, and Bill Venners. *Programming in Scala. Artima*, second edition, 2010.

[Ode14] Odersky, Martin. *The Scala Language Specification, Version* 2.8. EPFL, January 2014. Available on the web at http://www.scalalang. org/docu/files/ScalaReference.pdf (accessed November 17, 2014).

[Ora] Oracle. *Java Platform, Standard Edition 8 API Specification*. Available on the web at https://docs.oracle.com/javase/8/docs/api/index.html (accessed November 17, 2014).

[Sob10] Sobral, Daniel. "Implicit tricks – the Type Class pattern." June 2010. Available on the web at http://dcsobral.blogspot.com/2010/06/ implicit-tricks-typeclass-pattern.html (accessed November 17, 2014).

[Spi] Spiewak, Daniel and David Copeland. *Scala Style Guide*. EPFL. Available on the web at http://docs.scala-lang.org/style/ (accessed November 17, 2014).

[Sue] Suereth, Josh. "Implicit Classes." Available on the web at http://docs.scala-lang.org/overviews/core/implicit-classes.html (accessed November 17, 2014).

[Why] "Why is my abstract or overridden val Article?" Available on the web at http://docs.scala-lang.org/tutorials/FAQ/initializationorder. html (accessed November 17, 2014).

[Zau10a] Zaugg, Jason. "In Scala, why can't I partially apply a function without explicitly specifying its argument types?" March 2010. Available on the web at http://stackoverflow.com/a/2394063/391960 (accessed November 17, 2014).

[Zau10b] Zaugg, Jason. "Why 'avoid method overloading'?" March 2010. Available on the web at http://stackoverflow.com/questions/2510108/ why-avoid-methodoverloading/# 2512001 (accessed November 17, 2014).

作者简介

Andrew Phillips

Andrew 专攻并发和高性能应用，在为多个不同国家的公司工作期间开发了大规模系统。Andrew 是较早的开源开发者和社区成员，曾参与 Multiverse，这是用于 AKKa 的原始的 STM 实现；他为 Apache jclouds 做出贡献，领导 Java 云库；共同维护 Scala 谜题 Web 站点。他定期为开发者站点写文章，还经常在大型会议和小型研讨会上发言。

Andrew 在爱丁堡大学研究人工智能和数学，他还对机器学习、量子计算和计算神经科学保持着浓厚的兴趣。

Nermin Šerifović

Nermin Šerifović 有超过 10 年用 Java 技术开发企业应用软件的经验。他的大部分职业生涯致力于构建后台平台。自从 2009 年以来，Nermin 开始痴迷 Scala，2011 年开始专业 Scala 实践。他是哈佛继续教育学院的教员，在这里讲授 Scala 并发编程课程，并在不同的会议上发言。

Nermin 还是活跃的 Scala 社区成员，组织了波士顿地区的 Scala 狂热用户组，还是东北 Scala 讨论会创始团队的一员。Nermin 是 Scala 谜题 Web 站点的共同创建者。

Nermin 拥有康奈尔大学的计算机专业的工程硕士学位，他的兴趣领域包括并发式分布式系统、响应式和函数式编程。

主题索引

Scala 谜题

让代码做我们希望它做的事，是一名开发者的基本目的。然而实际上，往往我们自认为已经理解的代码，表现出的行为却和我们的预期相反，这类情况就显得有趣而且很重要了。

本书中的谜题，正是基于这样的场景而衍生出来的，本书收集了众多具有谜题特点的 Scala 代码示例。这不仅可以用一种寓教于乐的方式更好地理解 Scala 这一富有表现力的语言，而且可以帮助 Scala 开发人员认识许多容易忽视的雷区和陷阱，从而避免导致系统中的 bug 并给开发者带来压力。

本书适合对 Scala 语言以及函数式编程感兴趣的程序员阅读。

异步社区
人民邮电出版社
www.epubit.com.cn

异步社区 www.epubit.com.cn
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

ISBN 978-7-115-46007-3

ISBN 978-7-115-46007-3
定价：49.00 元

分类建议：计算机/程序设计/Scala
人民邮电出版社网址：www.ptpress.com.cn